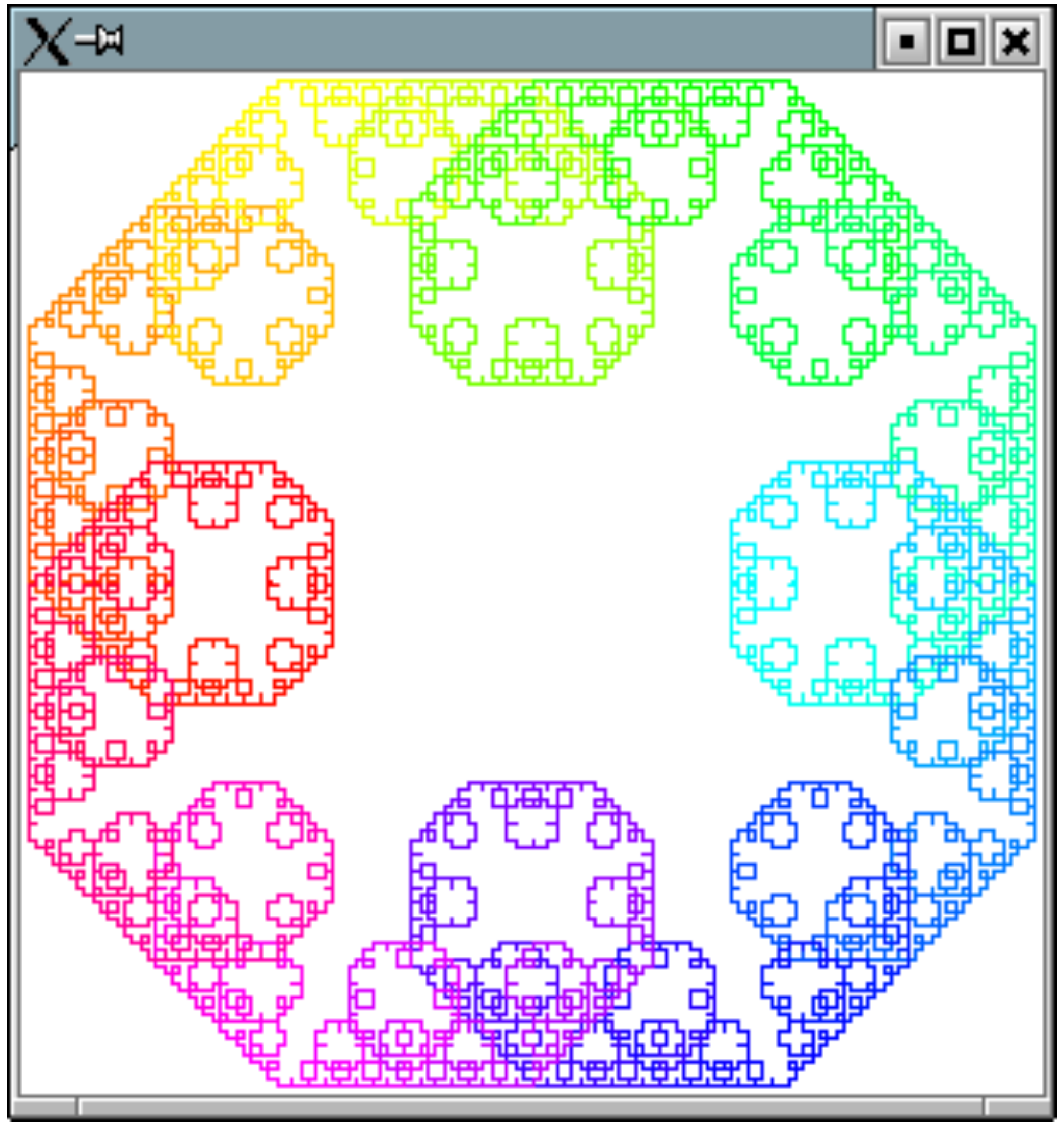
ＷＷＬ採択高校の１・２年生を対象とした
名古屋大学附属高等学校・教養教育院共催
「高大接続探究セミナー」

Pythonでフラクタルを描画しよう

アドバンスコース



名古屋大学　山里敬也

yamazato@nagoya-u.jp

スケジュール

日時	ベーシック	アドバンス
7月22日（月） 10：30〜12：00 講義	Google Colabortory (Python)入門 Turtle Graphics入門	Google Colabortory (Python)入門 Turtle Graphics入門
7月22日（月） 13：00〜14：30 演習	Turtle Graphics で 自分のイニシャルを描こう	Turtle Graphics で多角形を描こう
7月23日（火） 10：30〜12：00 講義	Turtle Graphics で多角形を描こう	再帰関数とフラクタル （コッホ曲線，シェルピンスキーのガスケット、2分木，Levy曲線，Drangon曲線）
7月23日（火） 13：00〜14：30 演習	Turtle Graphics で絵を描こう	Turtle Graphics で フラクタルを描こう

資料について

日時	ベーシック	アドバンス
7月22日（月） 10：30〜12：00 講義	Google Colabortory (Python)入門 Turtle_Graphics_Basic.ipynb を使います	Google Colabortory (Python)入門 Turtle Graphics入門 Turtle_Graphics_Basic.ipynb を使います
7月22日（月） 13：00〜14：30 演習	自分のイニシャルを描こう	Turtle Graphics で多角形を描こう
7月23日（火） 10：30〜12：00 講義	Turtle Graphics で多角形を描こう Turtle_Graphics_Basic.ipynb を使います	再帰関数とフラクタル （コッホ曲線、シェルピンスキーのガスケッ Turtle_Graphics_Advanced.ipynb を使います
7月23日（火） 13：00〜14：30 演習	Turtle Graphics で絵を描こう	Turtle Graphics で フラクタルを描こう

関数を定義しよう

def 関数名（引数1，引数2, …）：

関数は「def」で宣言

関数が受け取る値：引数

関数の結果（返値）は「return」で指定

$$c = multiply(a, b)$$

$$multiply(a, b) = a * b$$

TABもしくはスペース4つ開ける

```
def multiply(a,b):
    c = a*b
    return c
# multiply(2,3) の結果をプリント
print("c=",multiply(2,3))
```

```
c= 6
```

pythonの関数名は**スネークケース**で書くことが多い

例：turtle_graphics

- 分かりやすい名前をつける
- 小文字で始まる
- 単語を組み合わせる場合アンダースコア（_）で繋げる

フロー制御

実際のプログラムでは，式の評価結果に基づき命令（処理）をスキップしたり，くり返したり，いくつかある命令（処理）の一つを実行したりすることができます.

このような処理のことをフロー制御と言います.

フロー制御はフローチャート（流れ図）で表すことができます.

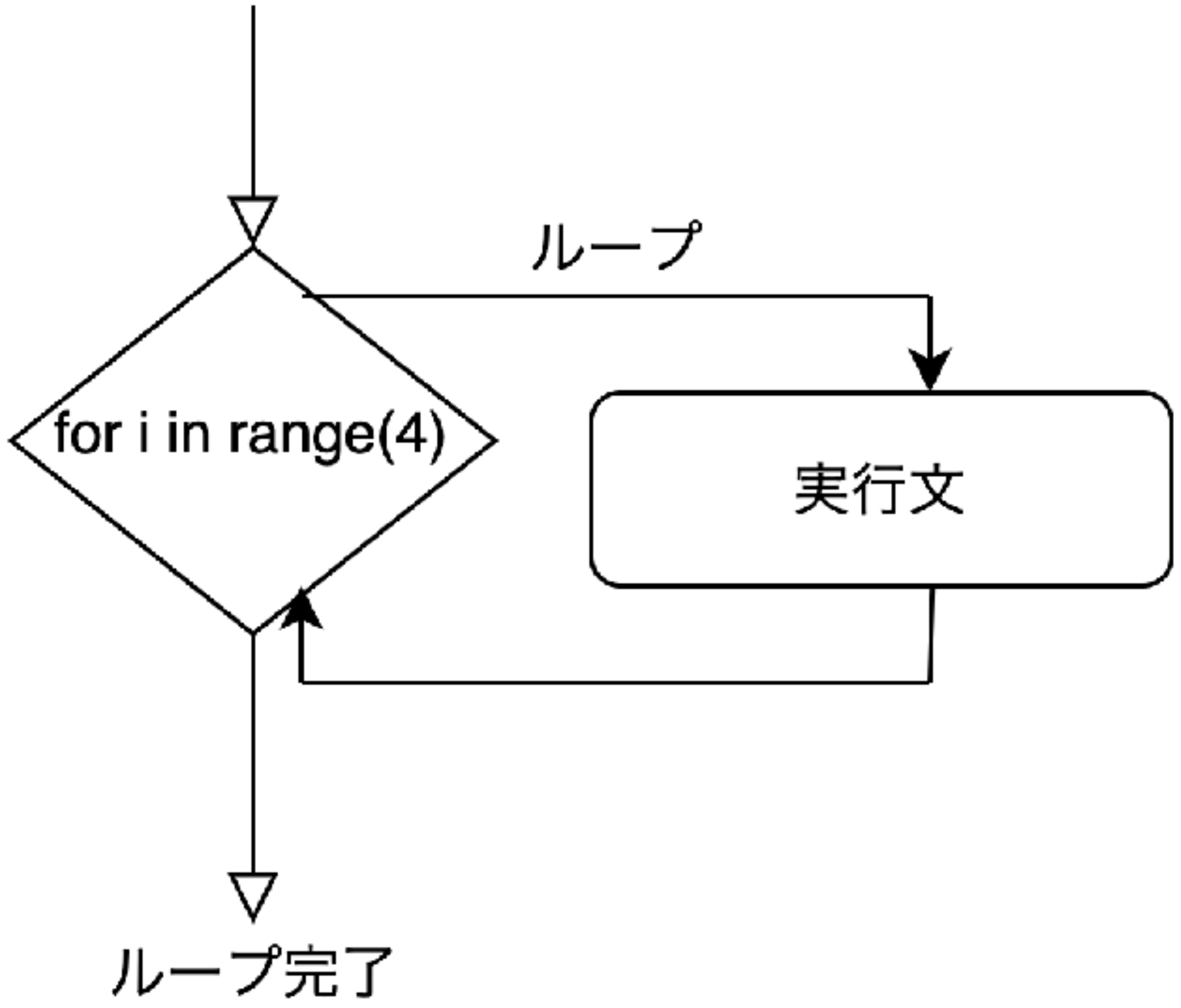
代表的なフロー制御には

- くり返し処理
  - for, while
- if（条件分岐）
  - if, elif, else

があります.

for ループと range

くり返し




```
for i in range(4):
    forward(100)
    right(90)
```

```
for 変数 in オブジェクト:
    実行文
```

文字列・リスト・関数など

`range()`

整数列のリストを返す関数

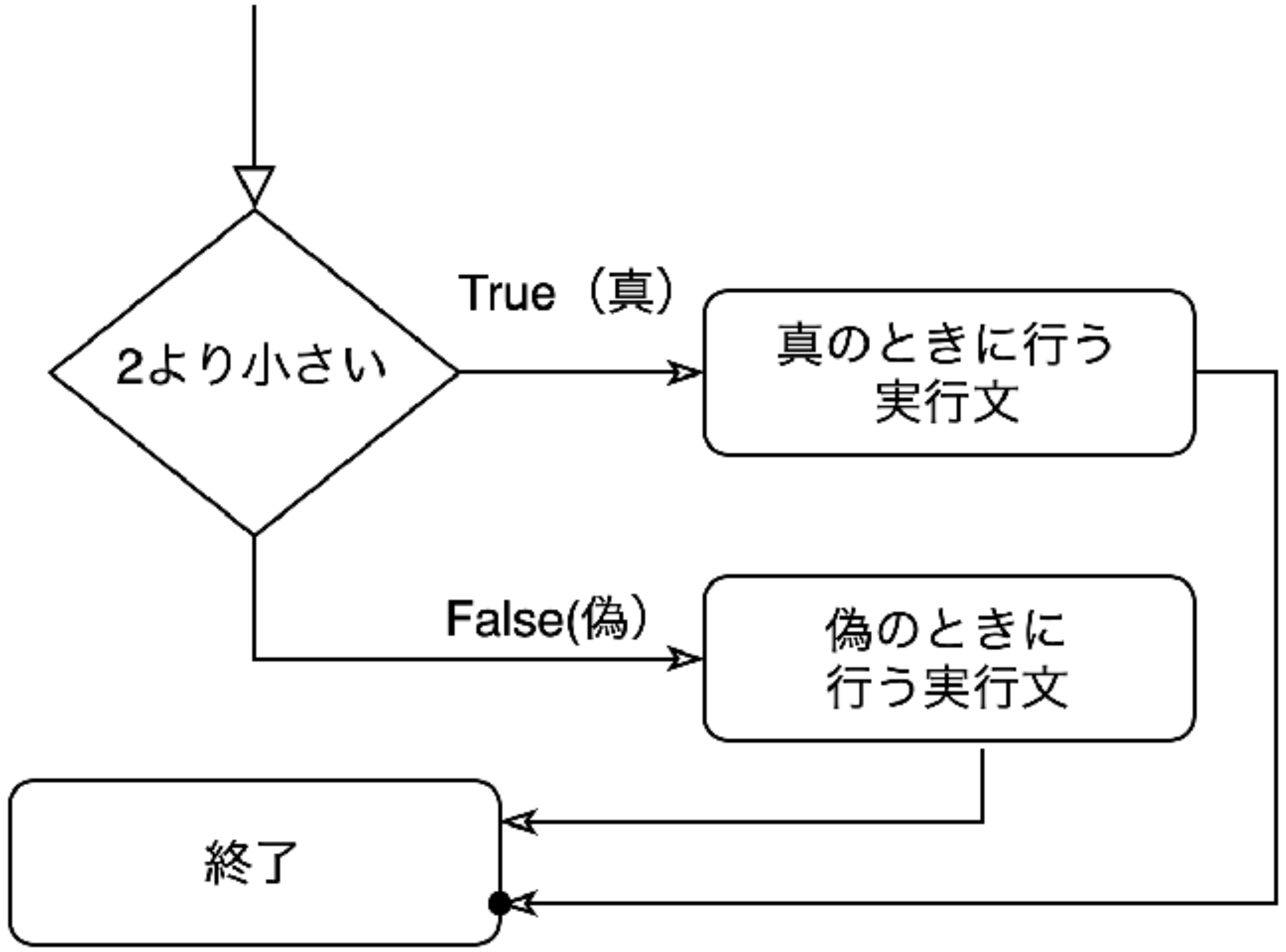
range(start, stop[, step])

range(0,4,1)
> 0, 1, 2, 3
range(4)
> 0, 1, 2, 3
range(0,4,2)
> 0, 2
range(4,0,-1)
> 3, 2, 1, 0

if と else

もし〜ならば，そうでなければ〜

真か偽を評価



if 条件式：

式が**真**のときに実行

else:

式が**偽**のときに実行

```
number = 2*3*4
if number < 2:
    print("Number is less than two.")
else:
    print("Number is not less than two.")
```

比較演算子

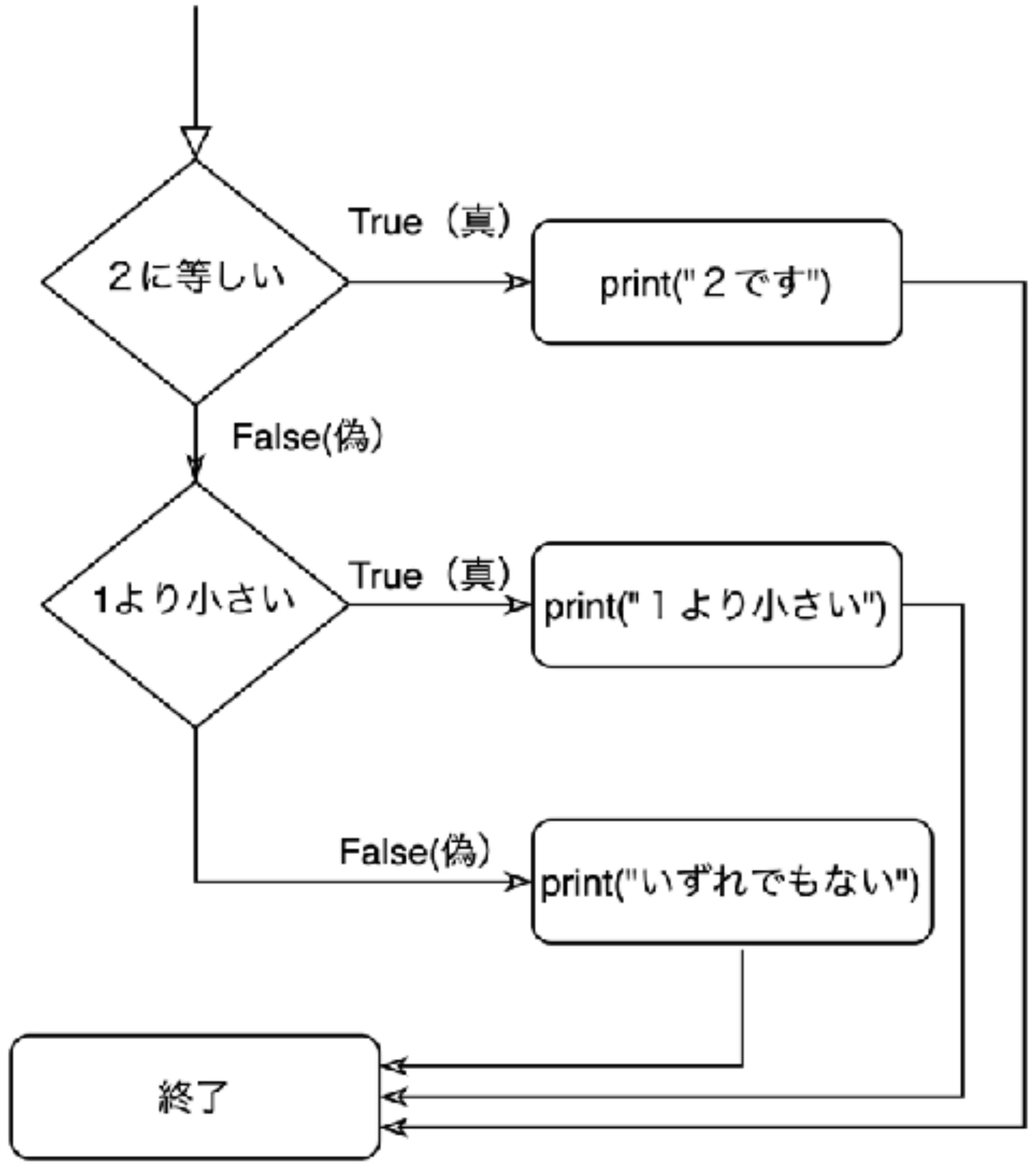
演算子	意味
==	等しい
!=	等しくない
>	より大きい
<	より小さい
>=	以上
<=	以下

ブール演算子

AND, OR, NOT

if, elif, else

もし〜ならば，そうでなくもし〜ならば，いずれでも無ければ〜




```
number = 2*3*4
if number == 2:
    print("2です")
elif number < 2:
    print("1より小さい")
else:
    print("いずれでもありません.")
```

いずれでもありません.

if 条件式1：

式1が**真**のときに実行

elif 条件式2:

式2が**真**のときに実行

else:

いずれでも無い場合に実行

比較演算子

演算子	意味
==	等しい
!=	等しくない
＞	より大きい
＜	より小さい
>=	以上
<=	以下

ブール演算子
AND, OR, NOT

再帰呼び出し

関数内で自分自身を呼び出す

　引数を与えないと、無限ループになる

再帰構造を持った関数の定義が容易に

　再帰呼び出しはループでも実現できるが，可読性は再帰呼び出しの方が良い

例：階乗，フィボナッチ数

$$n! = \prod_{k=1}^{n} k = n \times (n-1) \times \cdots \times 3 \times 2 \times 1$$

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n \times (n-1)!, & \text{if } n > 0 \end{cases}$$

終了条件

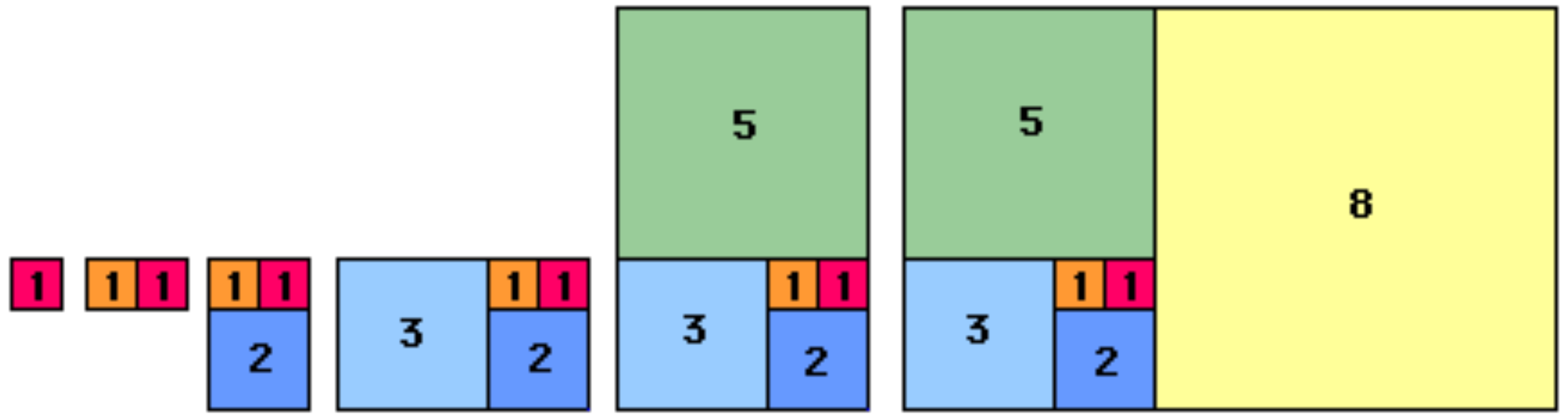
自分自身を呼び出す

```
def factor1(n):
# 再帰呼び出しによる階乗計算
    if n == 0:
        return 1
    else:
        return n * factor1(n - 1)

def factor2(n):
# for 文により階乗計算
    answer = 1
    for i in range(1, n+1):
        answer = answer *i
    return answer
```

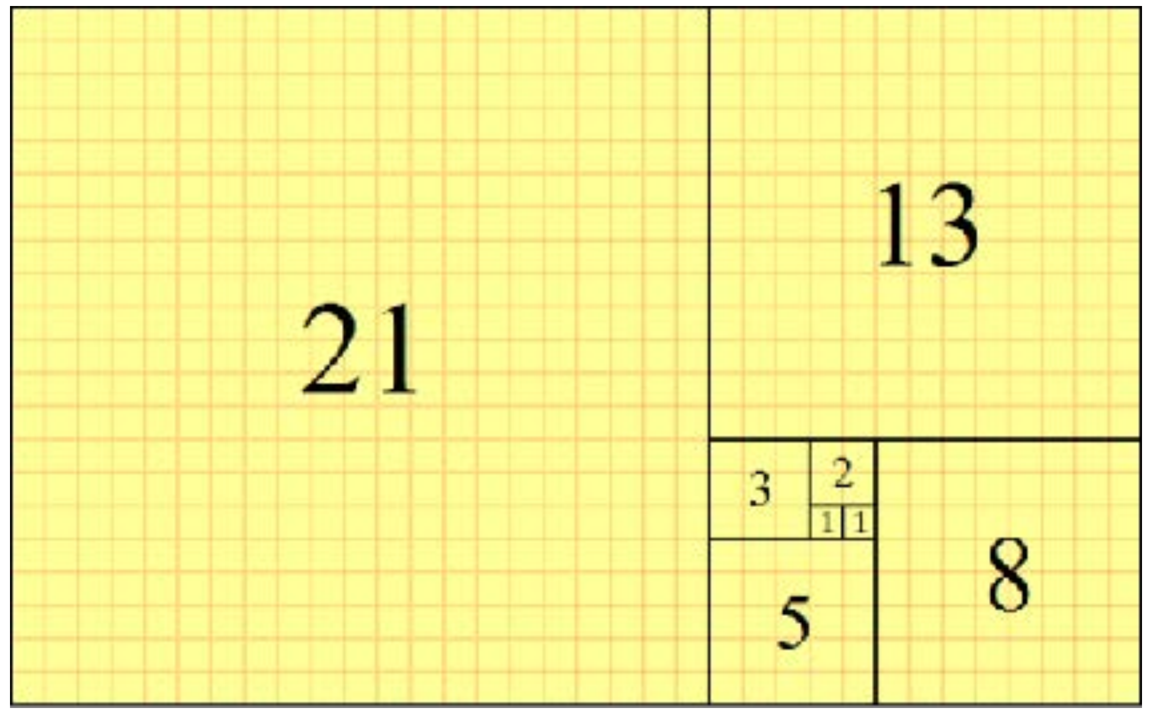
フィボナッチ数

フィボナッチ数列が生み出す螺旋は、世界で最も美しい螺旋

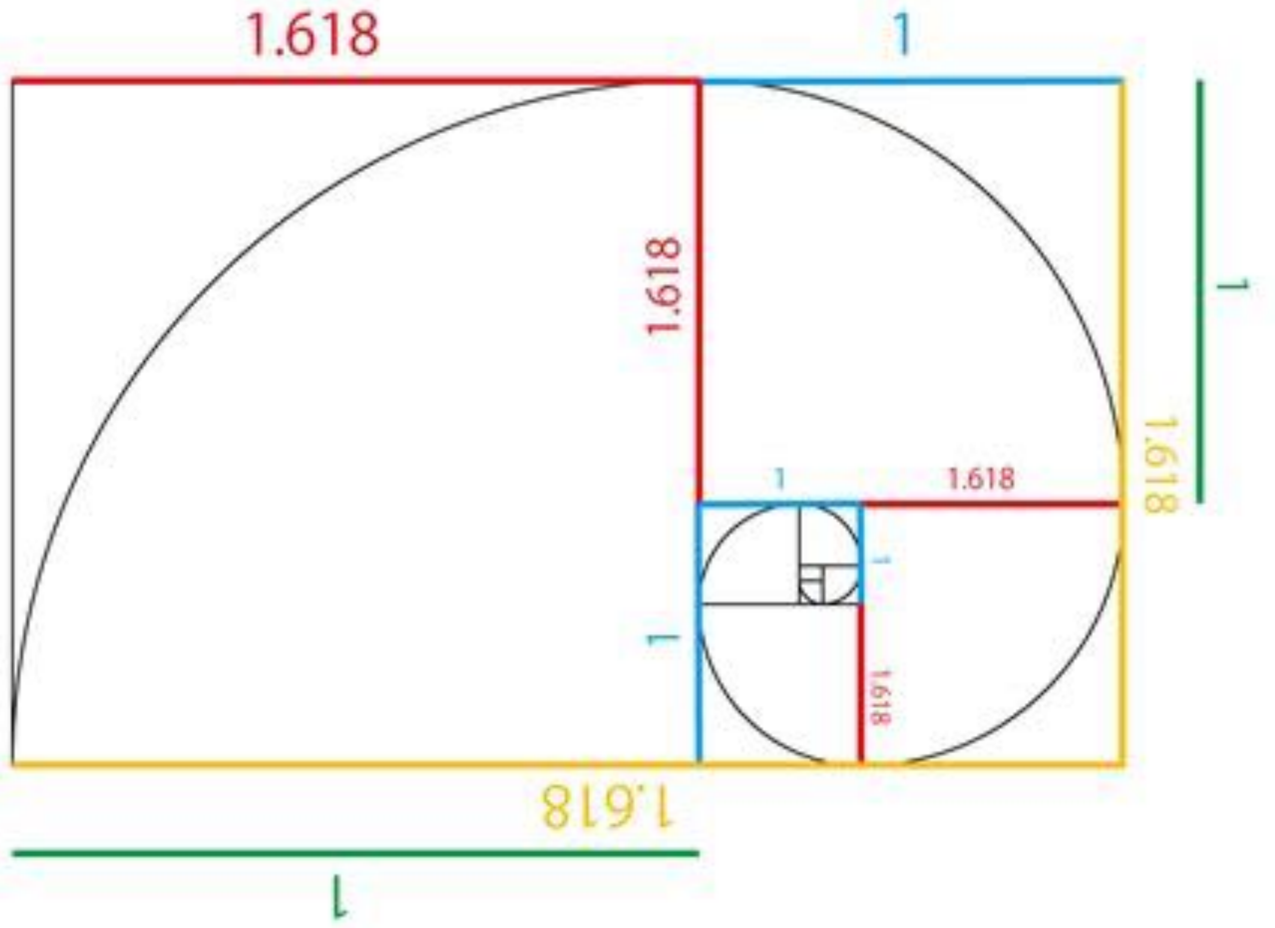
黄金比は「1 ：(1+√5)÷2」=1.618…
これはフィボナッチ数列の隣り合う数字の比と一致します。

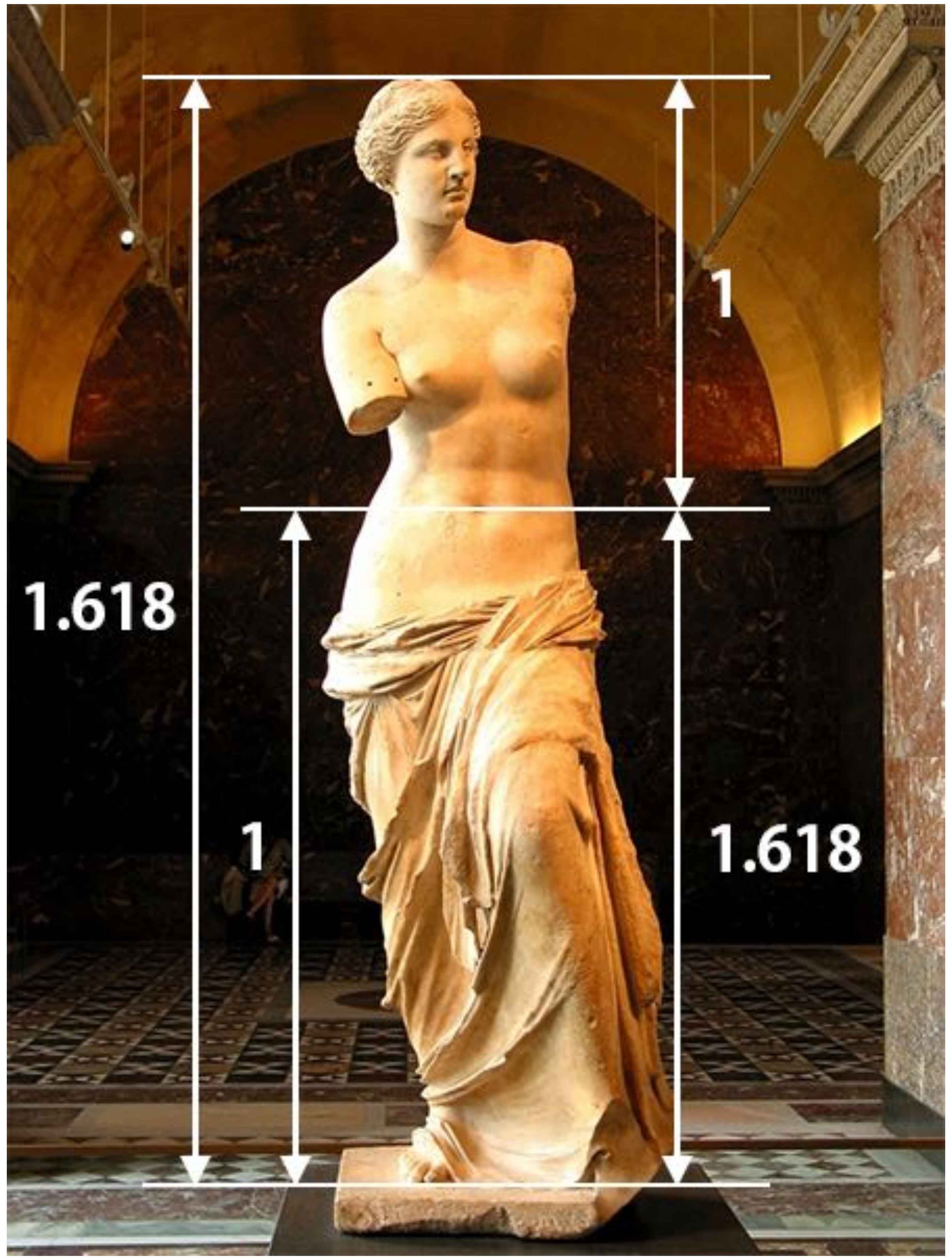


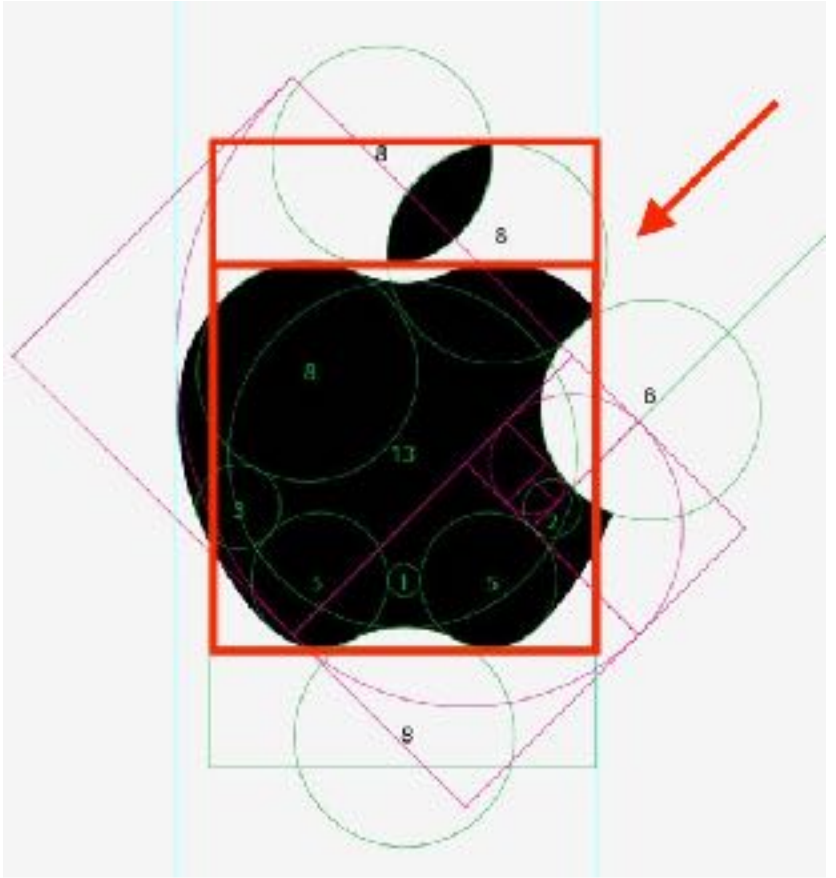
FIBONACCI SQUARES

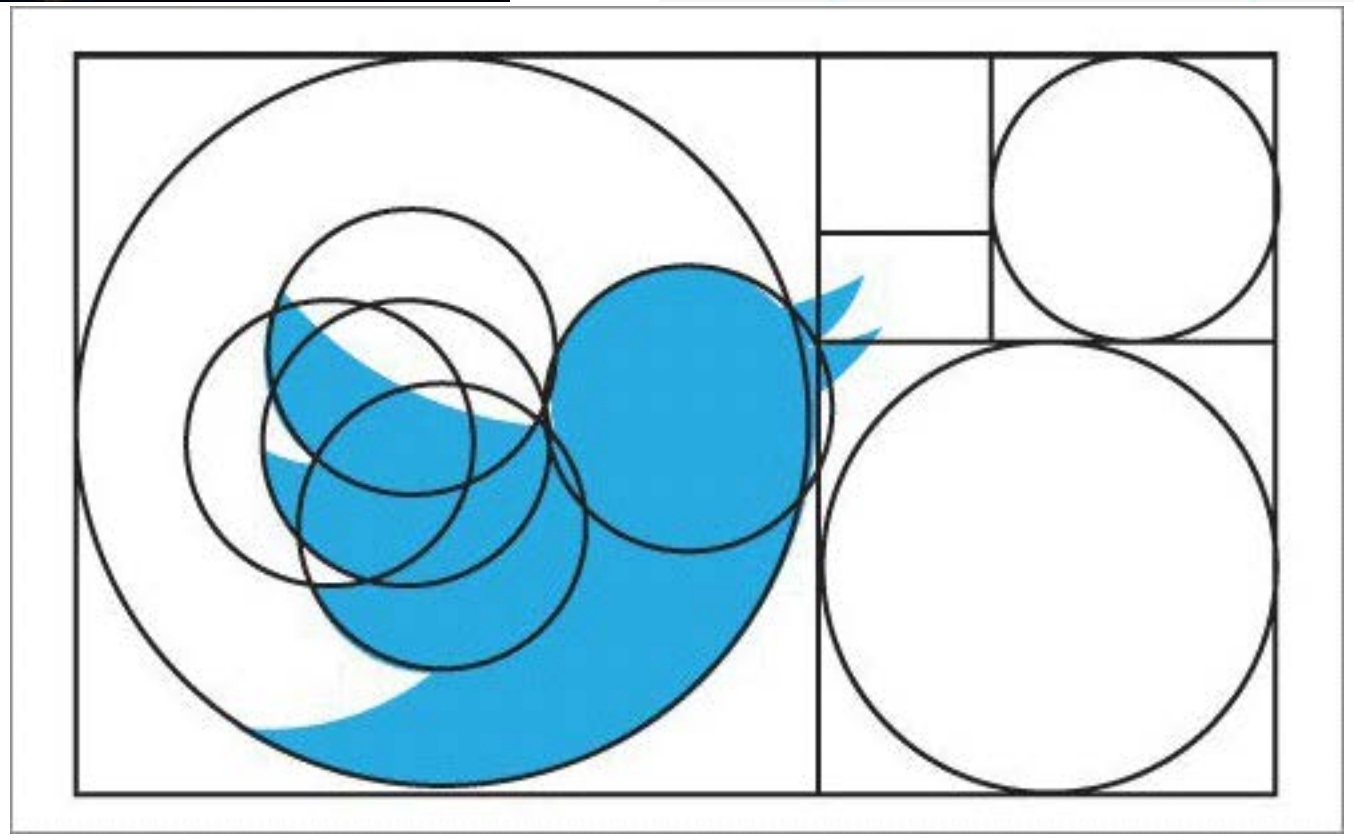


黄金比　1 : 1.618









階乗

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n \times (n-1)!, & \text{if } n > 0 \end{cases}$$

```
def factor1(n):
# 再帰呼び出しによる階乗計算
    if n == 0:
        return 1
    else:
        return n * factor1(n - 1)

def factor2(n):
# for 文により階乗計算
    answer = 1
    for i in range(1, n+1):
        answer = answer *i
    return answer
```

フィボナッチ数

$$F_0 = 0,\ F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \quad (n \geq 2)$$

```
def Fibonacci1(n):
# 再帰呼び出しによるフィボナッチ数列
    if n < 2:
        return n
    else:
        return Fibonacci1(n-1) + Fibonacci1(n-2)

def Fibonacci2(n):
# for文によるフィボナッチ数列
    if n<2:
        return n
    else:
        a=1
        b=1
        for i in range(n-2):
            total = a + b
            b=a
            a= total
        return a
```

亀さんを再帰的に動かしてフラクタルを描かせよう

タートルグラフィックスで再帰的に描画してみましょう.

だんだんと小さくなるような図形の描画にチャレンジします.

具体的には，引数をもった関数の再帰を利用して描画していきます.

フラクタルとは、自己相似性という性質を持っている図形で、代表的なものに、マンデルブロ集合というのがあります。（検索してみよう）

コッホ曲線

コッホ曲線 はフラクタル図形の一つで，線分を3等分し、分割した2点を頂点とする正三角形の作図を無限に繰り返すことによって描くことができます.

1次（くり返し数1）のコッホ曲線

これを基本図形として，各辺を基本図形に置き換えます.
2次（くり返し数2）のコッホ曲線は以下になります.

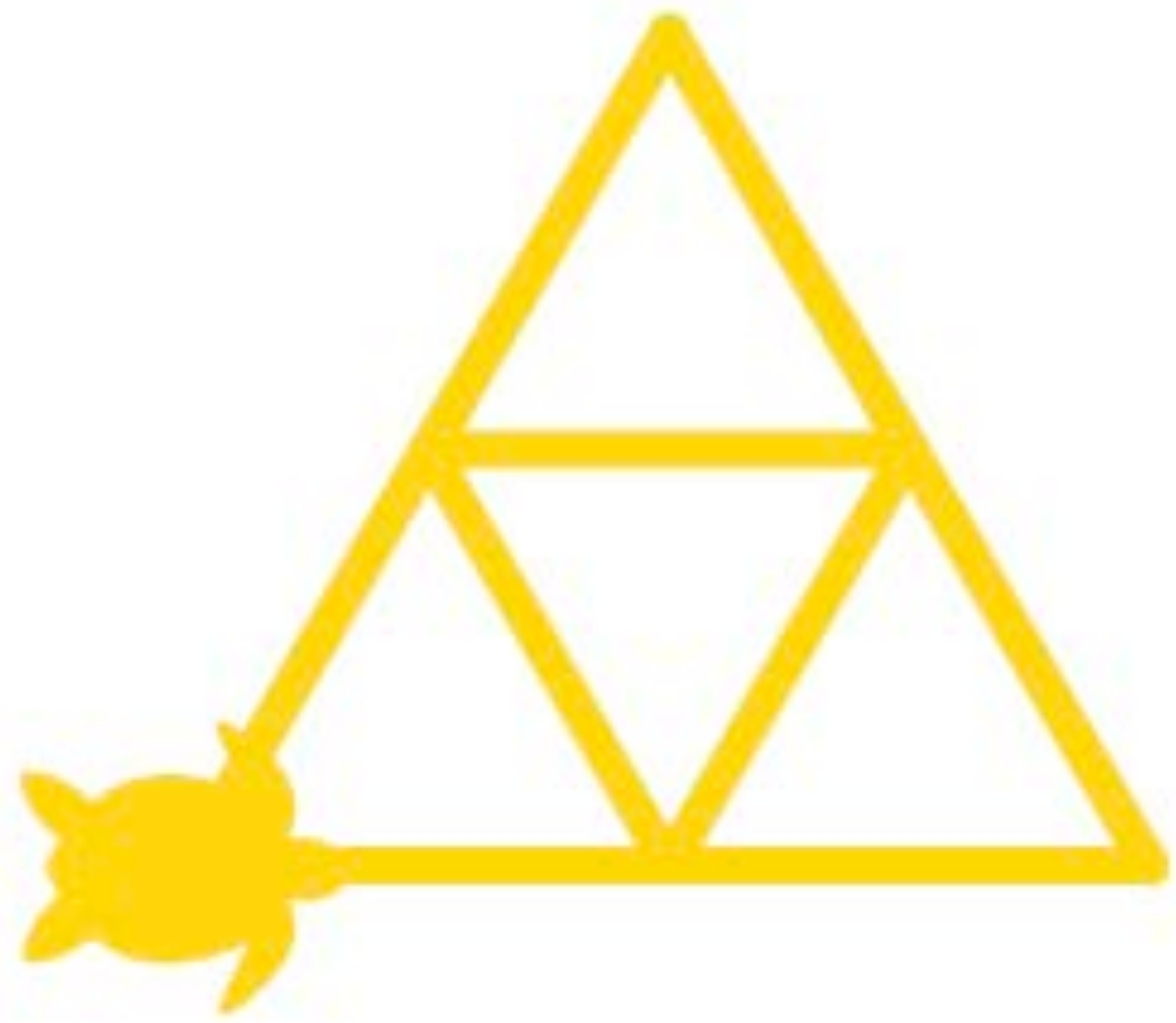
以上をくり返すことでコッホ曲線を描くことができます.

```python
from ColabTurtle.Turtle import *
initializeTurtle()
def koch(n, length):
    if n <= 0:
        forward(length)
    else:
        koch(n-1, length/3)
        left(60)
        koch(n-1, length/3)
        right(120)
        koch(n-1, length/3)
        left(60)
        koch(n-1, length/3)
if __name__ == '__main__':
    speed(10)
    bgcolor("white")
    color("orange")
    penup()
    goto(250, 100) #亀さんを移動

    pendown()
    right(90) # 横線を引くために亀の向きを右へ

# コッホ曲線を4回転させる

    for i in range(4):
        koch(3, 300)
        right(90)
    penup()
    home()
```

シェルピンスキーのガスケット

- 正三角形が基本図形

正三角形の各辺の中心を結んで,
内側に正三角形を作ります



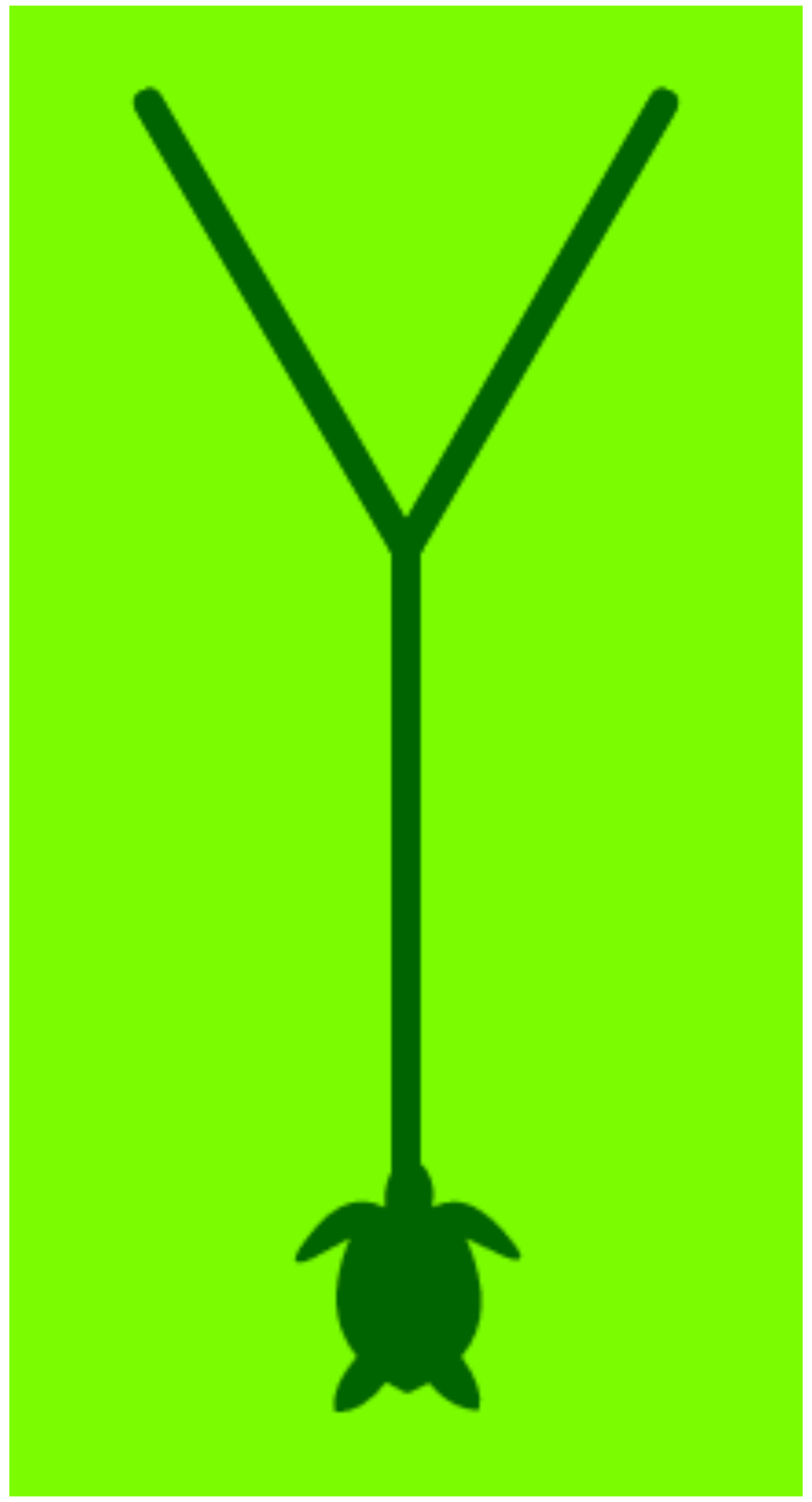
以上をくり返すことで描画していきます

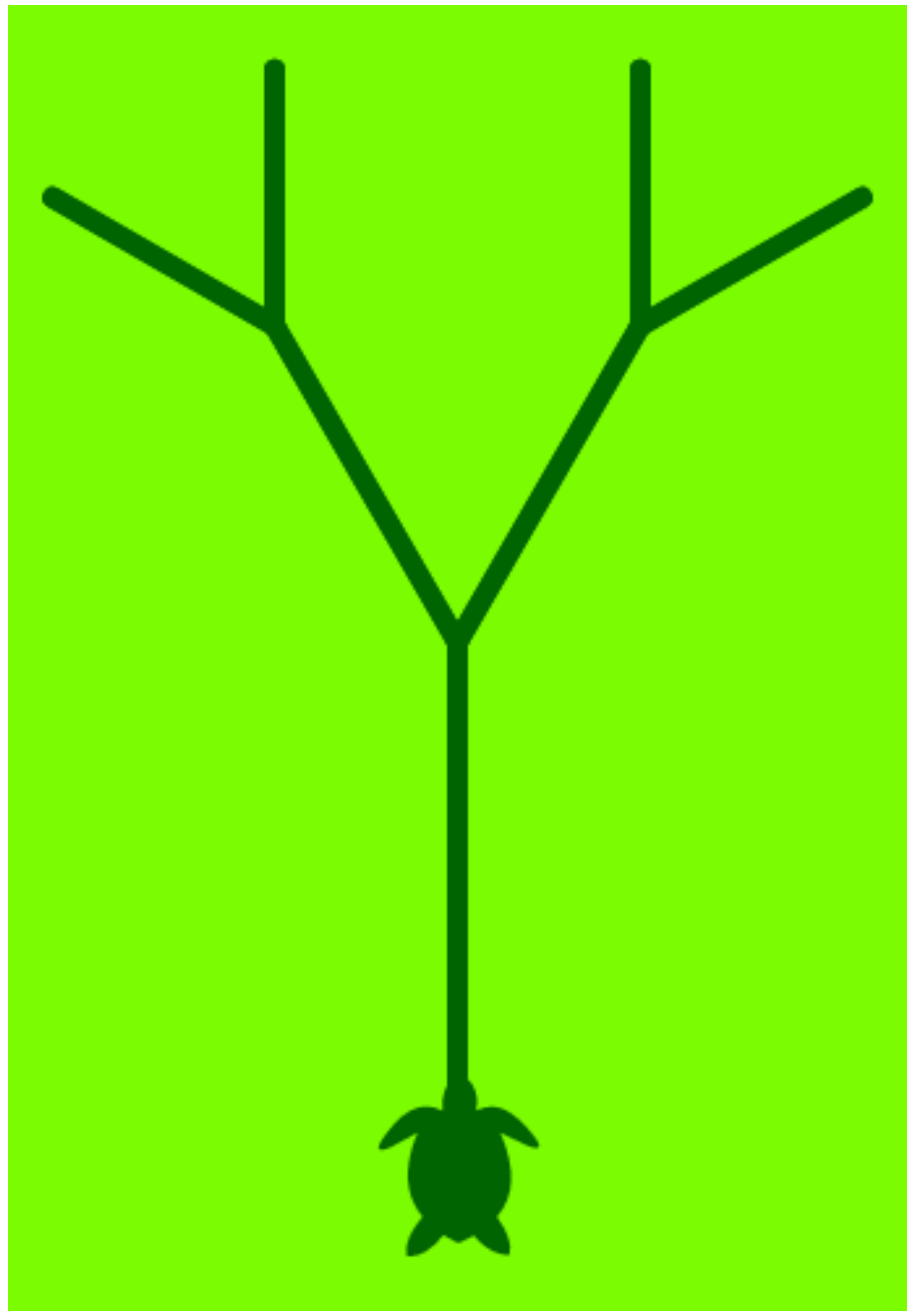
```python
from ColabTurtle.Turtle import *
initializeTurtle()

def Sierpinski_gasket(n, length):
    if n <= 0:
        # ここを考えよう

    else:
        Sierpinski_gasket(n-1, length/2) # 一辺を半分に

        forward(length) # 直線を描く

        left(120) # 120度左に

        Sierpinski_gasket(n-1, length/2)
        forward(length)
        left(120)
        Sierpinski_gasket(n-1, length/2)
        forward(length)
        left(120)

if __name__ == '__main__':
    speed(13)
    bgcolor("DarkRed")
    color("Gold")
    penup()
    goto(200,400) #亀さんを移動

    pendown()
    right(90) # 横線を引くために亀の向きを右へ

    Sierpinski_gasket(5, 400)
    penup()
    home()
```

二分木

- 1つの幹と2つの枝が基本図

次の次数（くり返し）では，この2つの枝を新たの幹として2つの枝を伸ばしていきます



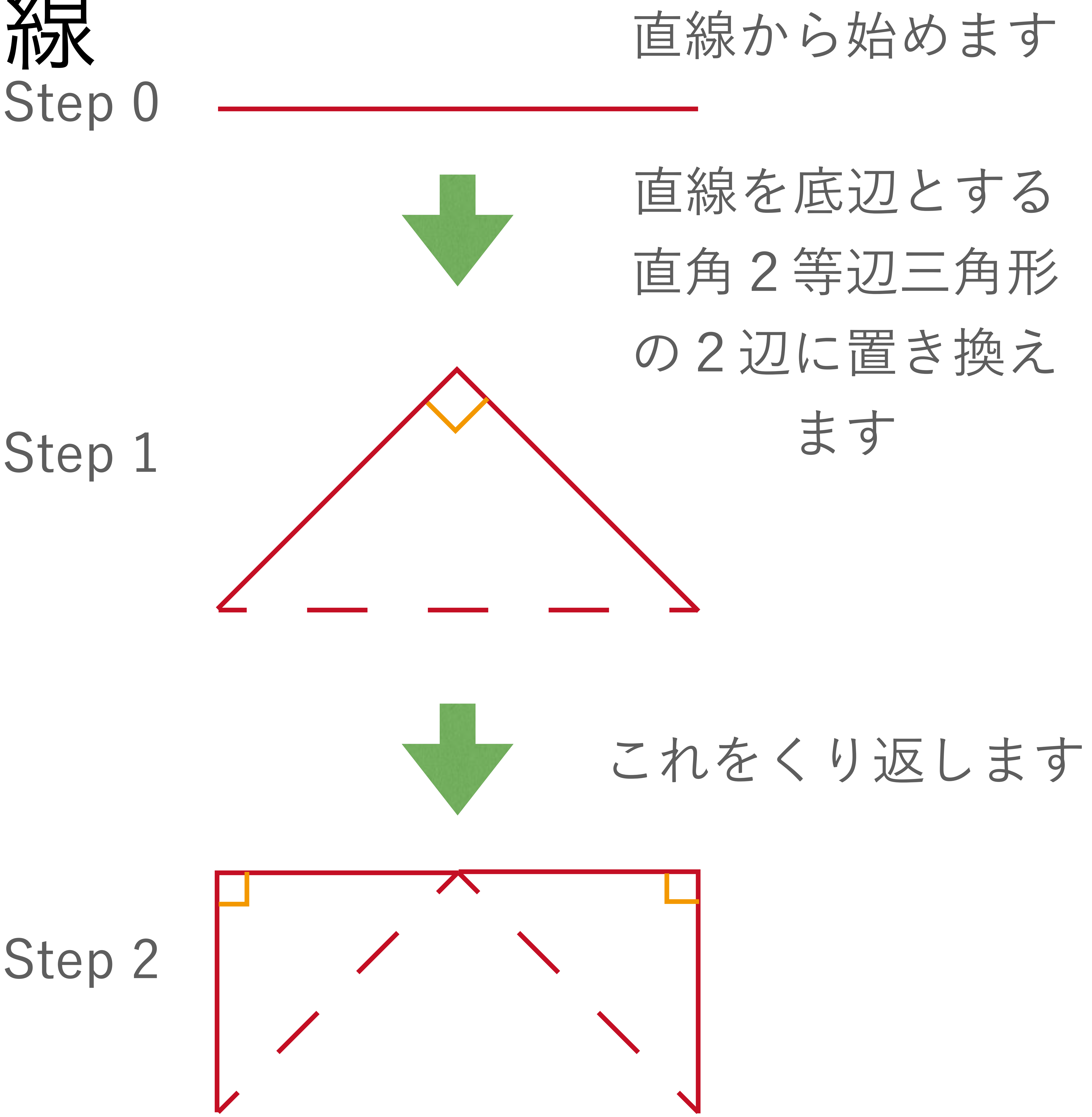


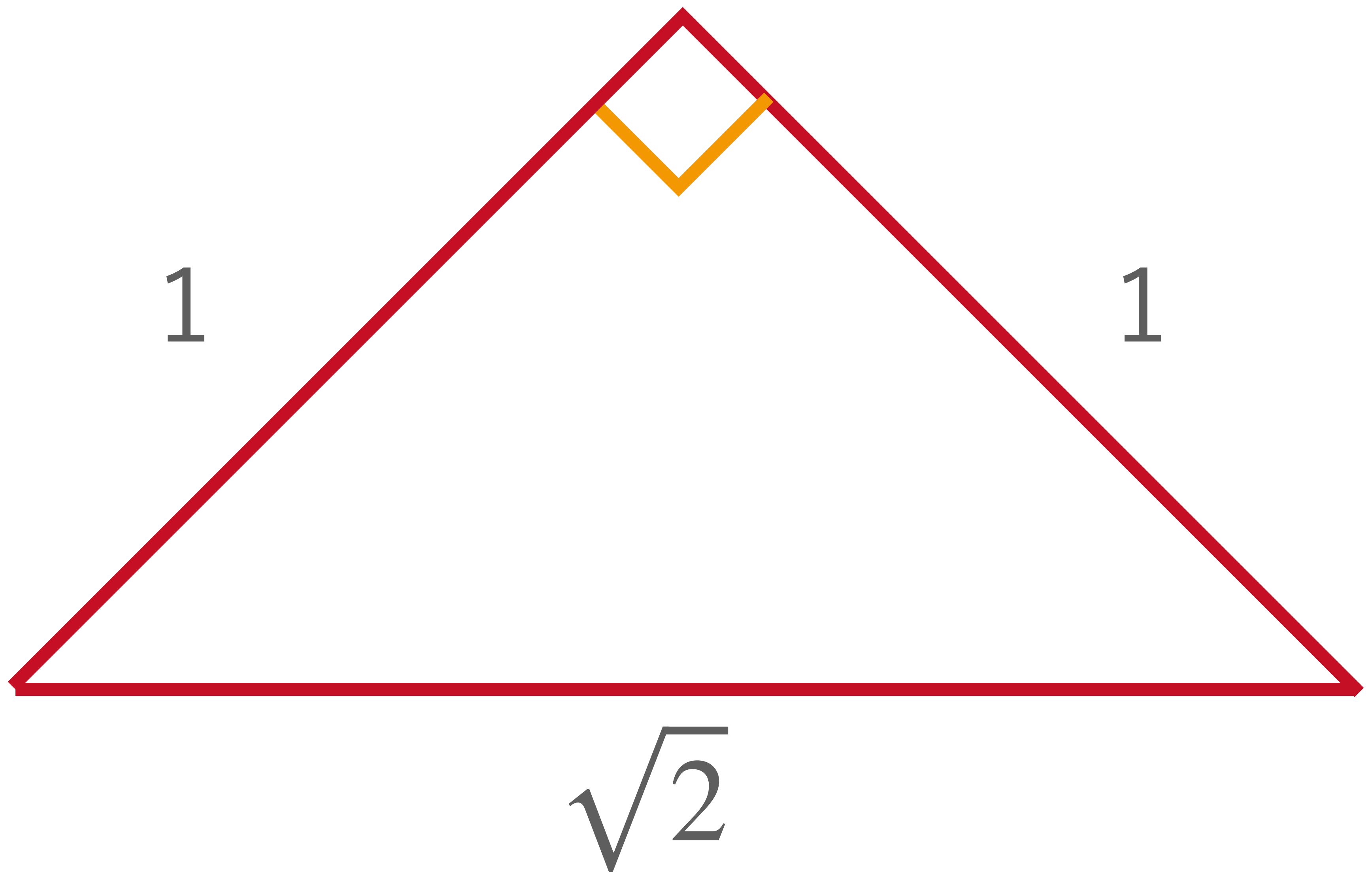
以上をくり返すことで枝を伸ばして木を描いていきます.

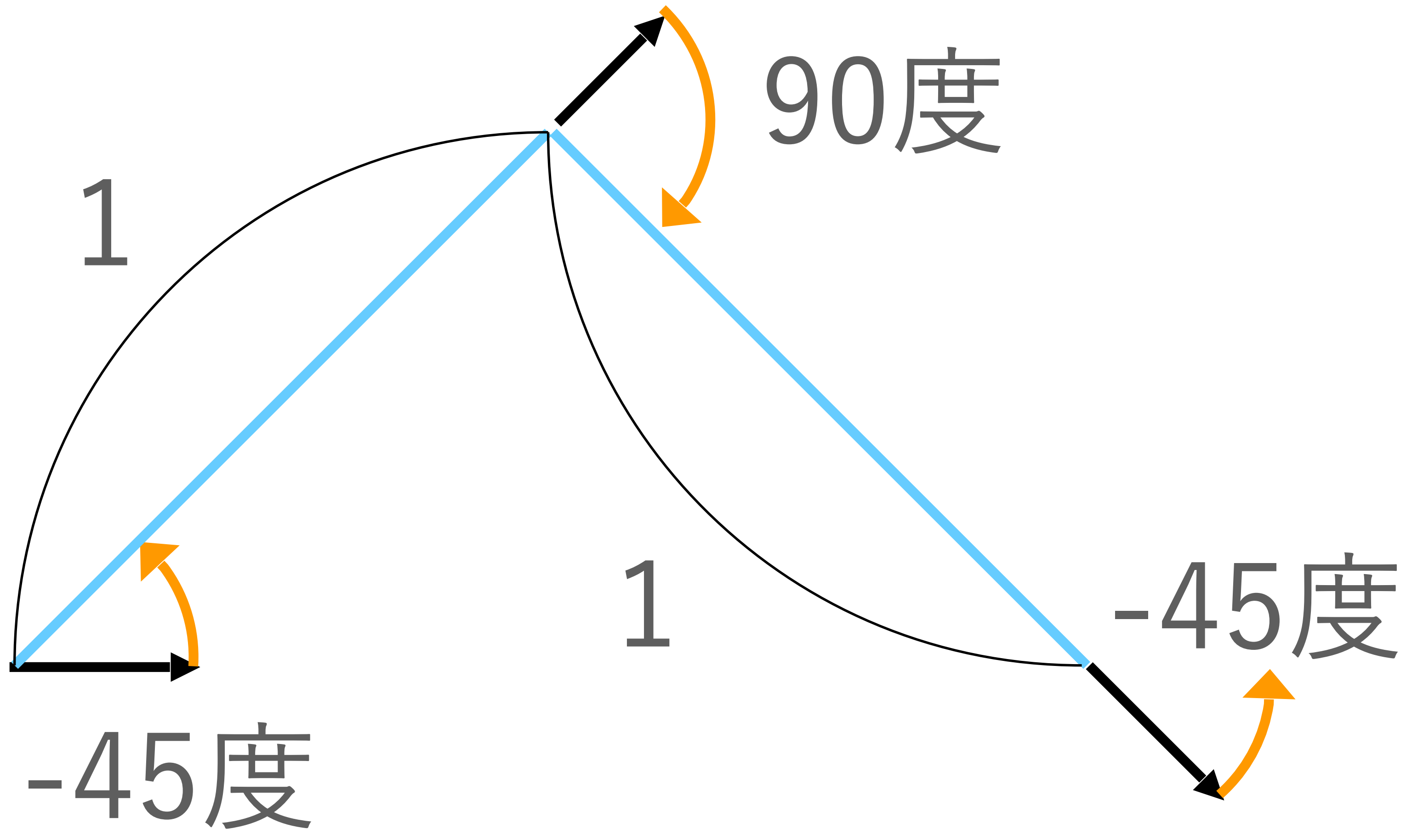
```python
from ColabTurtle.Turtle import *
initializeTurtle()

def binary_tree(n, length, angle):
    if n > 0:
        forward(length)
        right(angle)
        binary_tree(n-1, length * 0.7, angle)
        left(angle * 2)
        binary_tree(n-1, length * 0.7, angle)
        right(angle)
        backward(length)

if __name__ == '__main__':
    speed(13)
    bgcolor("LawnGreen")
    color("DarkGreen")
    penup()
    goto(400,400) #亀さんを移動

    pendown()
    binary_tree(8, 100, 30)
```

試してみよう：Levy曲線

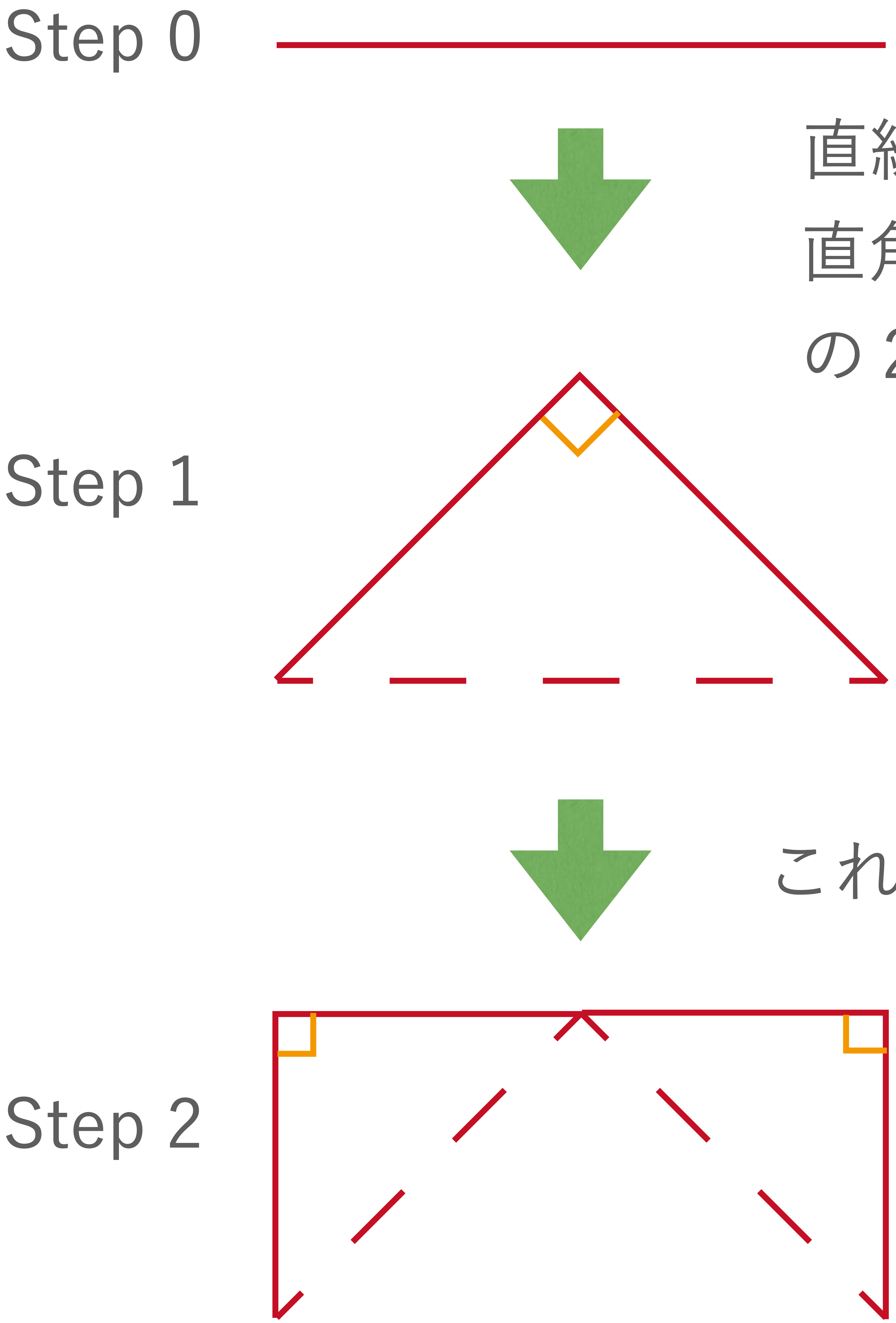
Ｌｅｖｙ曲線はフラクタルの一種。







直線から始めます



直線を底辺とする直角２等辺三角形の２辺に置き換えます

これをくり返します

length: 一辺の長さ

関数 **levy(length,n)** を考える

n: ステップ

length = sqrt(2)
n = 1 の場合

亀さんは右90度に進む

Step 0　$\mathrm{levy}(\sqrt{2},0)$

forward($\sqrt{2}$)

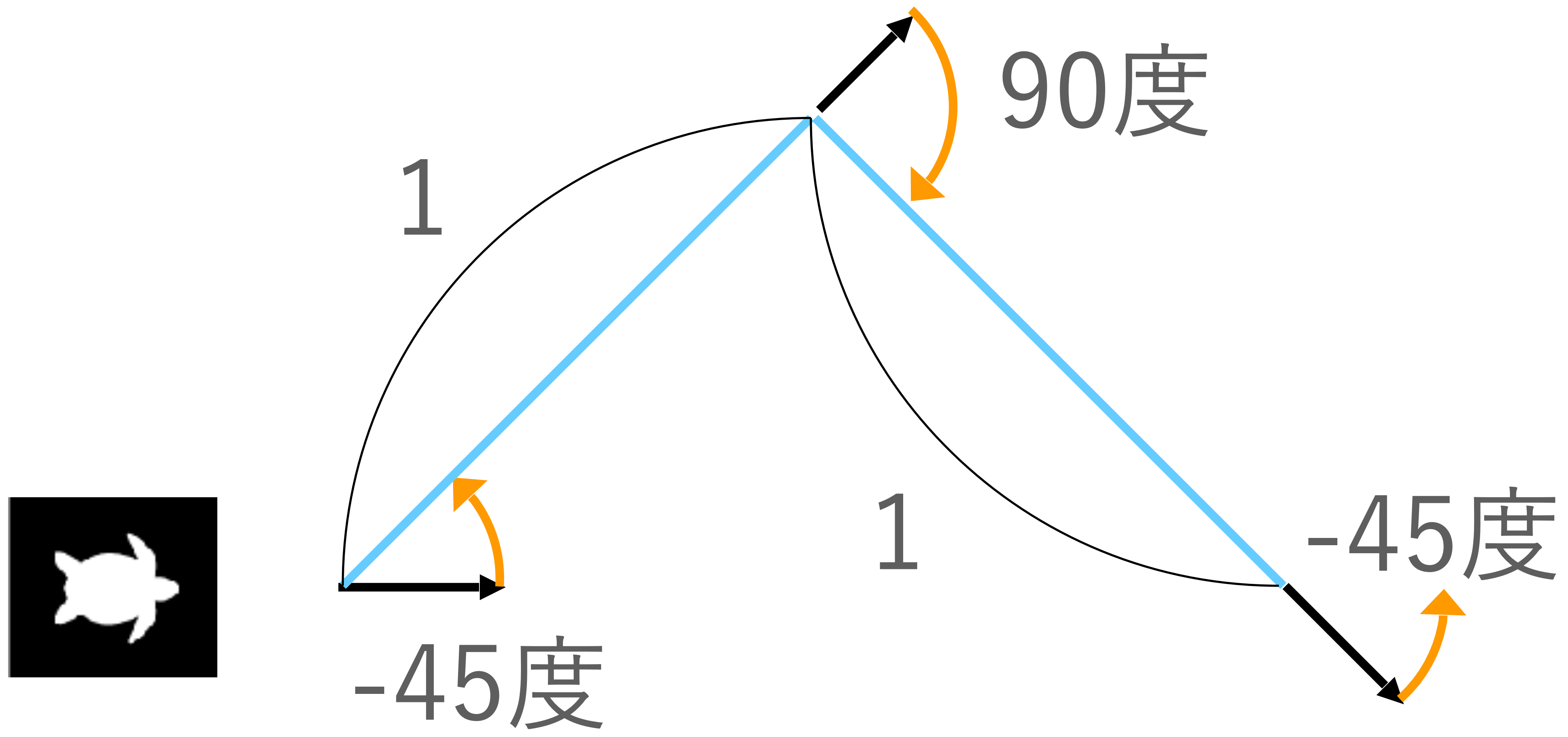
length: 一辺の長さ

関数 **levy(length,n)** を考える

n: ステップ

length = sqrt(2)
n = 1 の場合

Step 1　　$\text{levy}(\sqrt{2},1)$



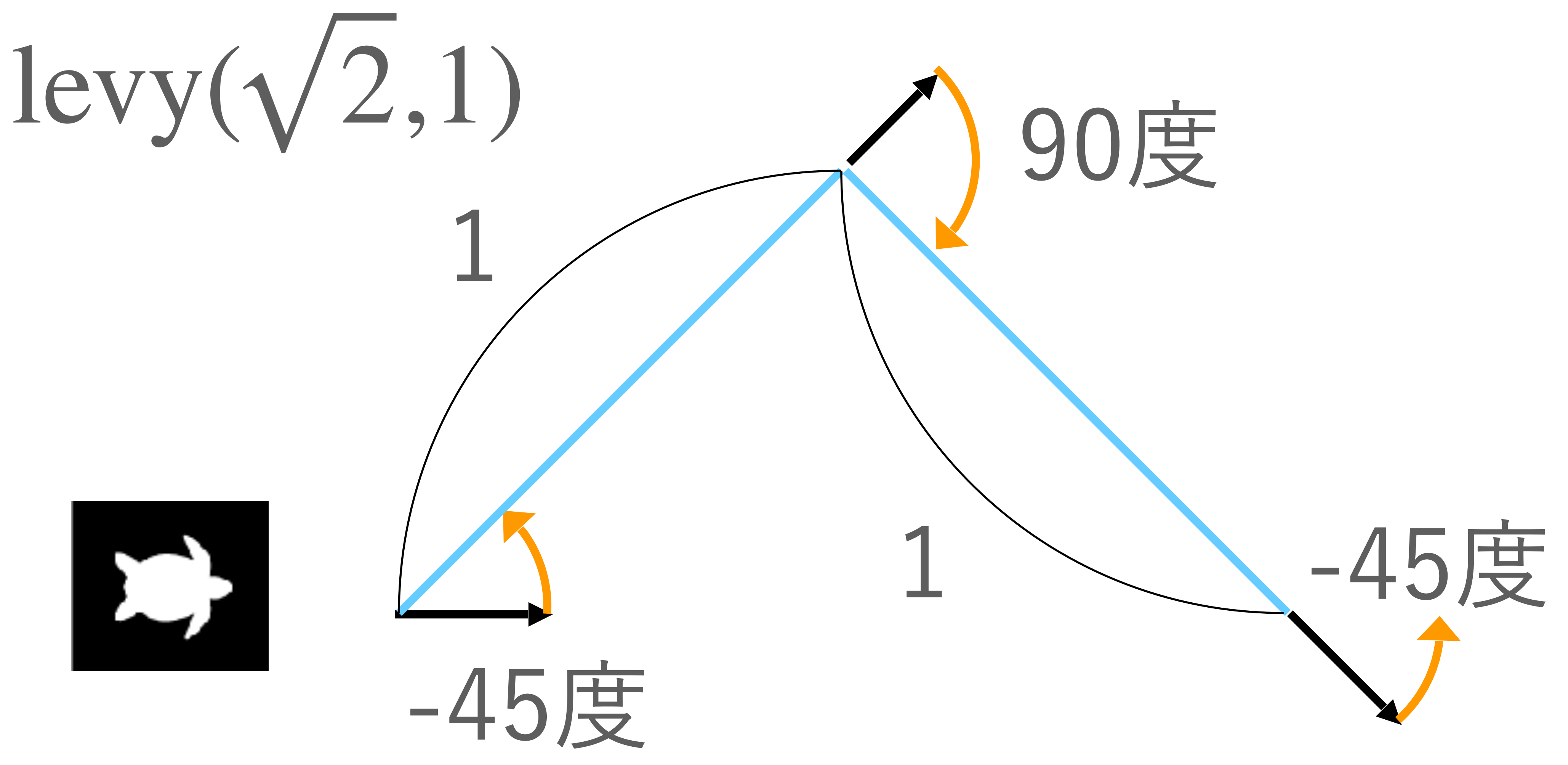
一辺の長さを$1/\sqrt{2}$倍

```
dl = length/sqrt(2)
left(45);
forward(1);   ➡  levy(1,0);
right(90);
forward(1);   ➡  levy(1,0);
left(45);
```

再帰呼び出し

再帰呼び出し

```python
def levy(length, n):
    if n == 0:
        forward(length)
    else:
        dl = length/sqrt(2)
        left(??)
        levy(dl,??)
        right(??)
        levy(dl,??)
        left(??)
```



length = sqrt(2)
n = 1 の場合

一辺の長さを$1/\sqrt{2}$倍

dl = length/sqrt(2)

left(45);

forward(1); → levy(1,0);

再帰呼び出し

right(90);

forward(1); → levy(1,0);

再帰呼び出し

left(45);

試してみよう：Dragon 曲線

直線をそれを底辺とする

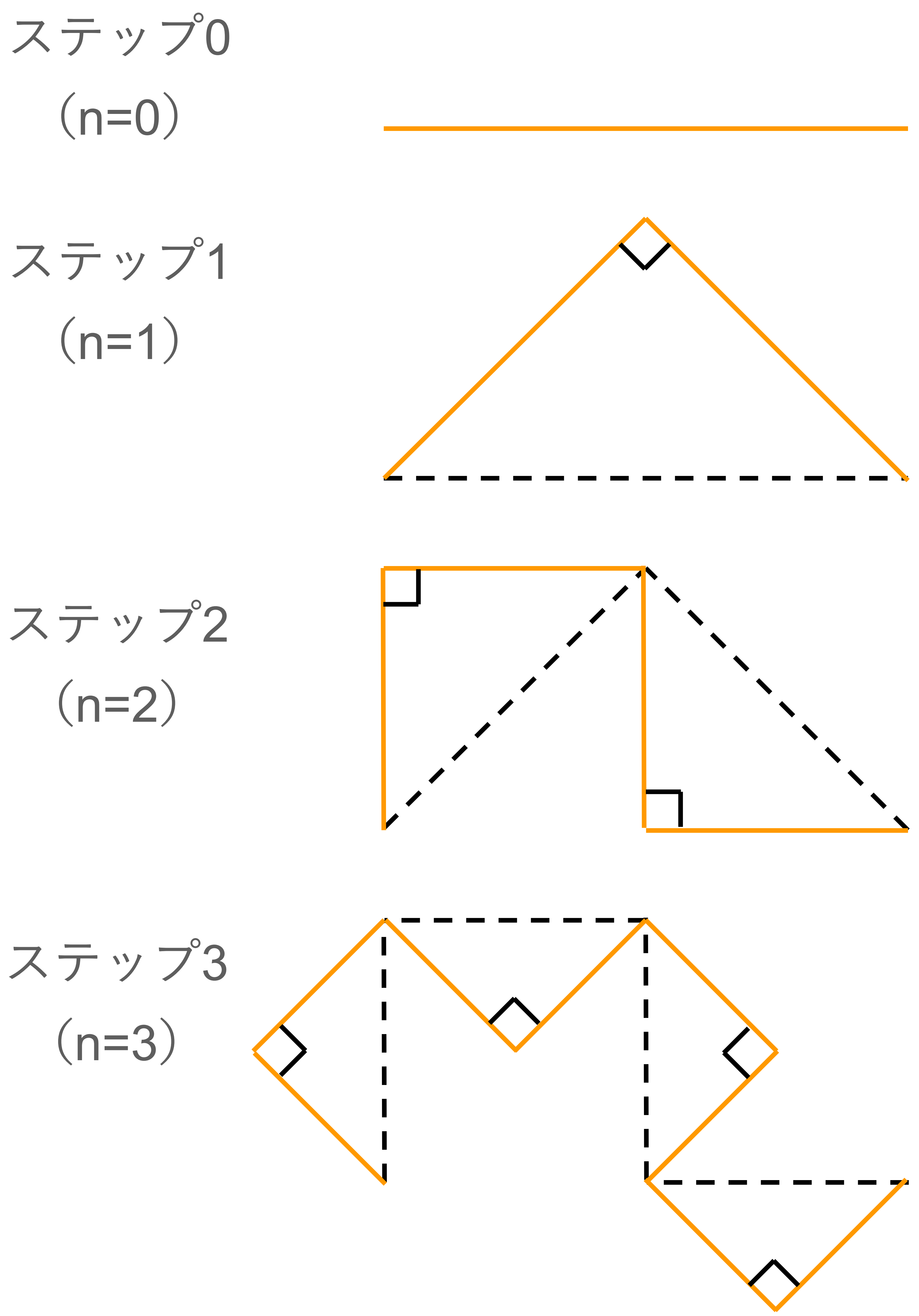
直角二等辺三角形の

２辺に置き換える

以降は、交互に上下に

入れ替わった辺で置き換える

これを繰り返す

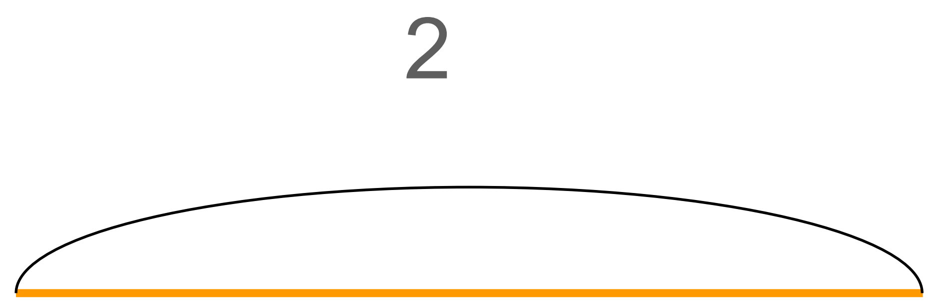


- n=8から12くらいがよいでしょう
- あまり n を大きくすると再帰処理に時間がかかって応答が遅くなります
- n の値をいろいろ変えた図を載せてもOKです

Dragon 曲線の描画のヒント

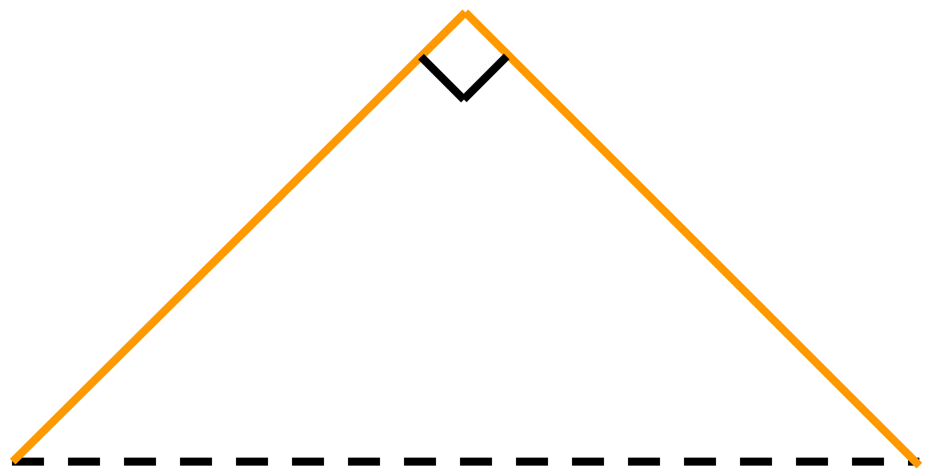
n=2,length=2 の Dragon 曲線

ステップ0
（n=0）



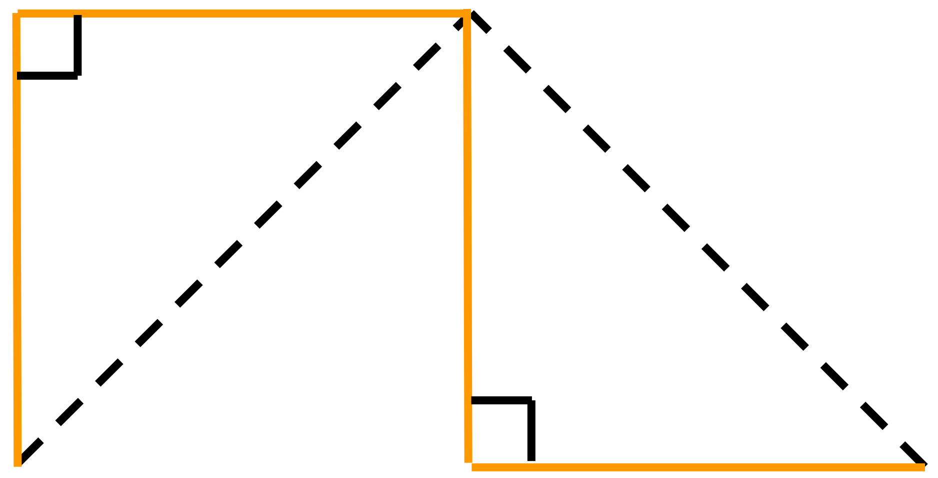

```
forward（2）；
```

ステップ1
（n=1）



```
left（45）；
forward（2/sqrt（2.0））；
right（90）；
forward（2/sqrt（2.0））；
left（45）；
```

ステップ2
（n=2）



```
left（45）；
  left（45）；
  forward（（2/sqrt（2.0））/sqrt（2.0））；
  right（90）；
  forward（（2/sqrt（2.0））/sqrt（2.0））；
  left（45）；
right（90）；
  left（-45）；
  forward（（2/sqrt（2.0））/sqrt（2.0））；
  right（-90）；
  forward（（2/sqrt（2.0））/sqrt（2.0））；
  left（-45）；
left（45）；
```

演習：フラクタルを描こう