CMPT 120
Intro to CS & Programming I

WEEK 3 (Jan. 20-24)

— *Jérémie O. Lumbroso* —

Lecture 8:

Some Introduction to Recursive Functions

http://www.sfu.ca/~jlumbros/Courses/CMPT120/

Notion central to many useful algorithms

RECURSIVE FUNCTIONS

Recursive Function

- Recursive function = function that calls itself
- For instance, **factorial function**

base case

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

recursive call

function definition

- A recursive function has generally two cases:
  - the base case, which ensures the function stops
  - the recursive case, which contains one (or several) recursive call(s)

Example

base case

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

recursive call

function definition

Example of a computation for 5!

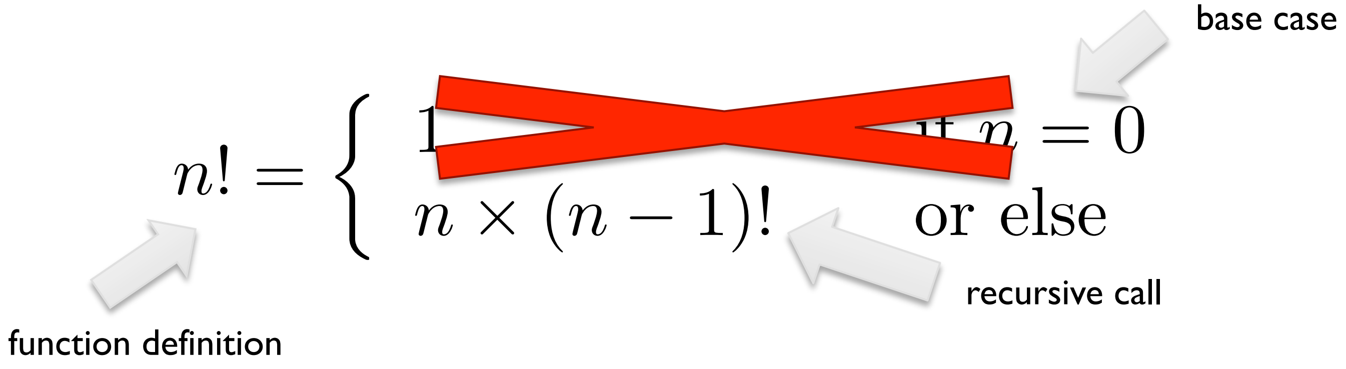
- 5! = 5 × (5-1)! = 5 × 4! (recursive case)
- 4! = 4 × (4-1)! = 4 × 3! "
- 3! = 3 × (3-1)! = 3 × 2! "
- 2! = 2 × (2-1)! = 2 × 1! "
- 1! = 1 × (1-1)! = 1 × 0! "
- 0! = 1 (base case)

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)
```

This yields in the end

- 5! = 5 × 4 × 3 × 2 × 1 × 1 = 125

What Happens With No Base Case?



base case

$$n! = \begin{cases} \xcancel{1 \qquad \text{if } n} = 0 \\ n \times (n-1)! \qquad \text{or else} \end{cases}$$

recursive call

function definition

- 5! = 5 × (5-1)! = 5 × 4! (recursive case)
- 4! = 4 × (4-1)! = 4 × 3! "
- 3! = 3 × (3-1)! = 3 × 2! "
- 2! = 2 × (2-1)! = 2 × 1! "
- 1! = 1 × (1-1)! = 1 × 0! "
- 0! = 0 × (0-1)! = 0 × (-1)! "
- (-1)! = (-1) × (-1-1)! = (-1) × (-2)! "
- (-2)! = (-2) × (-2-1)! = (-2) × (-3)! "
- …
- (-98332) = (-98332) × (-98332-1)! = (-98332) × (-98333)! "
- …
- (-19239323188)! = (-19239323188) × (-19239323188-1)! = (-19239323188) × (-19239323189)! "
- forever!

What About With a **BAD** Base Case?

base case

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

recursive call

function definition

- 5! = 5 × (5-1)! = 5 × 4! (recursive case)
- 4! = 4 × (4-1)! = 4 × 3! "
- 3! = 3 × (3-1)! = 3 × 2! "
- 2! = 2 × (2-1)! = 2 × 1! "
- 1! = 1 × (1-1)! = 1 × 0! "
- 0! = 0 (**bad** base case)

This yields in the end

- 5! = 5 × 4 × 3 × 2 × 1 × 0 = 0, not the correct result!

Can We Mess Up the Recursive Call?

base case

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n + 1)! & \text{or else} \end{cases}$$

recursive call

function definition

- 5! = 5 × (5+1)! = 5 × 6! (recursive case)
- 6! = 6 × (6+1)! = 6 × 7! "
- 7! = 7 × (7+1)! = 7 × 8! "
- 8! = 8 × (8+1)! = 8 × 9! "
- 9! = 9 × (9+1)! = 9 × 10! "
- …
- 12383! = 12383 × (12383+1)! = 12383 × 12384! "

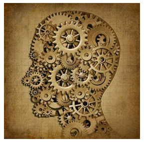
This also goes on forever.

Summary of Recursive Functions

- Recursive functions are what we call functions that need to call themselves
- Idea is that we compute the result of a function for a large parameter by computing it first for a smaller parameter
- For instance, we compute factorial(n-1) before we can compute factorial(n)
- The body of a recursive function contains an if statement with two cases
  - one **base case** in which we give fixed value and in which we **do not make a recursive call**
  - one **recursive case** in which we call the function itself for a **strictly decreasing value of the parameters**
- The base case, and the fact that the parameters are decreasing are both important properties to ensure that the function does not run forever

Understanding

How do you feel about recursive functions?



- **A** I knew about them fine before, this is not new for me
- **B** I don't think this is confusing me, it seems like a natural notion
- **C** Recursive functions are confusing, I need another example
- **D** This went too fast, I don't understand anything
- **E** I am in class because allergic to the sun outside

Fibonacci Sequence

- Another example of recursive function

base case

$$a_n = \begin{cases} 1 & \text{if } n \leqslant 1 \\ a_{n-1} + a_{n-2} & \text{or else} \end{cases}$$

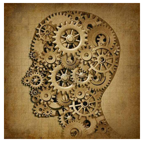
recursive call

function definition

- This translates in Python to

```
def fibonacci(n):
  if n <= 1:
    return 1
  else
    return fibonacci(n-1)+fibonacci(n-2)
```

Exercise



- Write a function to calculate $a^n = a \times a \times \ldots \times a$
- **Without using `**`** (exponentiation), only the math operations `*` (multiplication) and – (subtraction)
- Questions to ask yourself before you write code
  - what is the base case? (when do we stop the function? what fixed value do we return there so that the function works?)
  - what is the recursive call? do I modify the parameter `a`? what about `n`? and if so how do I modify `n`?
- Once those questions are answered you can fill in this code

```
def my_exponentiation(a, n):
  if ____________:
    return __________     # base case
  else:
    return __________     # recursive case that must contain call to my_exponentiation
```

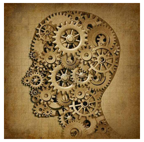
A Figured it out!!

B I am giving up, I don't understand at all

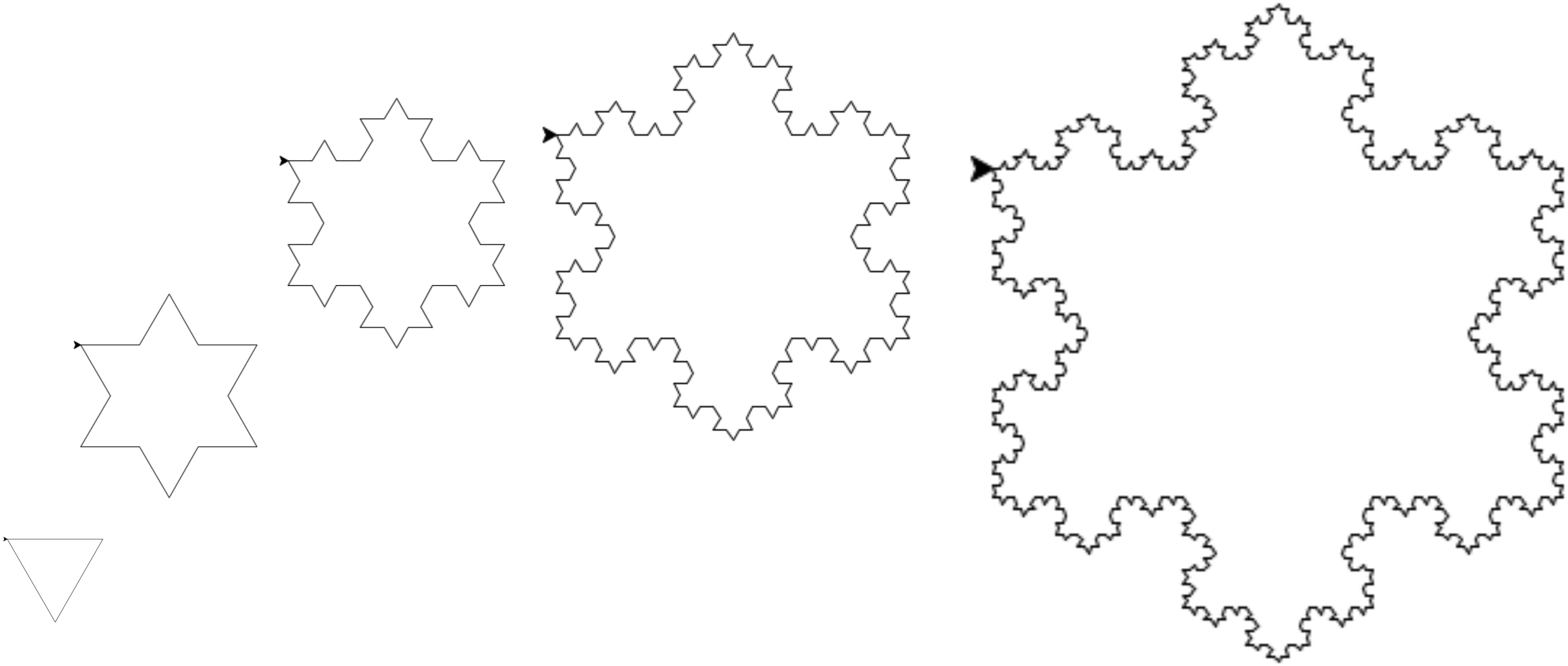
Solution

```
def my_exponentiation(a, n):
  if n == 0:
    return 1    # base case
  else:
    return a * my_exponentiation(a, n-1)    # recursive case
```

Did you get it right?



- **A** Yes, I got it right!
- **B** No, I did not get it right, but I see how I could have done it
- **C** No, I did not get it right, and I don't think I could do it



Fractals?

RECURSIVE TURTLES

Turtle Functions

- http://docs.python.org/2/library/turtle.html
- Short module name: `import turtle as t` (if you don't want to do `turtle.blahblah`, but `t.blahblah`)

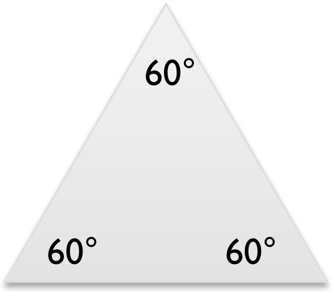
- Turtle movement
  - t.forward(length) or t.fd(length)
  - t.backward(length) or t.bk(length) or t.back(length)
  - t.right(angle) or t.rt(angle)
  - t.left(angle) or t.lt(angle)
  - t.setposition(x, y) to go to a specific position or t.home() to go to center
- Pen control (whether drawing or not)
  - t.pendown() or t.pd() or t.down()
  - t.penup() or t.pu() or t.up()
- Color control: t.pencolor(…) and t.fillcolor(…)
- Other functions
  - t.begin_fill() and t.end_fill() to fill a shape
  - t.clear() to erase screen without resetting
  - t.reset() to erase screen + center turtle

Koch Snowflake

- Write a line function that goes `length` in one direction

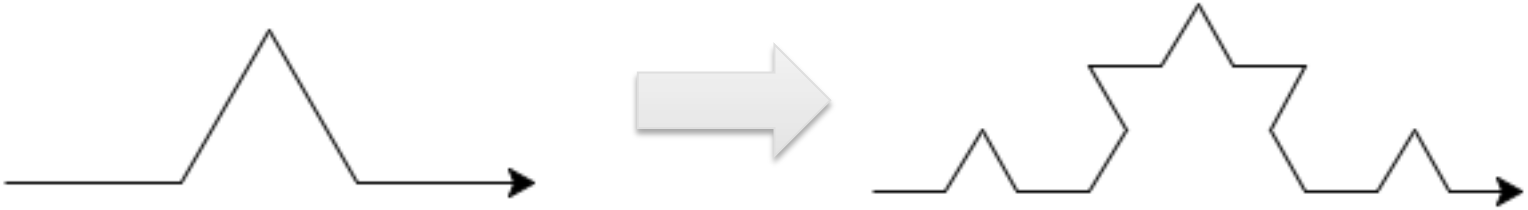
```
def normal_line(length):
  turtle.forward(length)
```

- Write a broken line function that cuts in one third the line and does an equilateral triangle



```
def draw_broken_line(length):
  draw_normal_line(length/3.)
  turtle.left(60)
  draw_normal_line(length/3.)
  turtle.right(120)
  draw_normal_line(length/3.)
  turtle.left(60)
  draw_normal_line(length/3.)
```

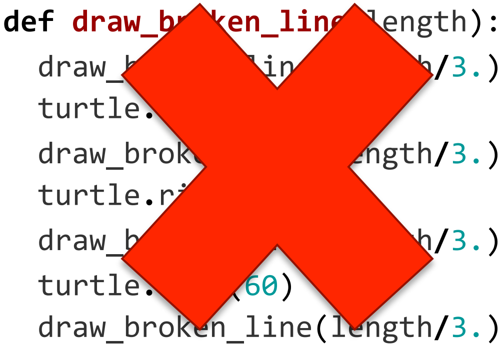
Recursive Line



- The broken-line (left) made up of normal lines
- Want the normal line segment of this broken-line to be replaced itself by a broken-line

```
def draw_broken_line(length):
  draw_normal_line(length/3.)
  turtle.left(60)
  draw_normal_line(length/3.)
  turtle.right(120)
  draw_normal_line(length/3.)
  turtle.left(60)
  draw_normal_line(length/3.)
```

???



```
def draw_br    ken_lin    length):
  draw_b         lin         h/3.)
  turtle.
  draw_brok               ength/3.)
  turtle.ri
  draw_b                     h/3.)
  turtle.        (60)
  draw_broken_line(length/3.)
```

We Need a Base Case

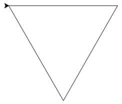
- This function is a good idea
- But does not work because it never stops (like when factorial without a base case just goes into negative numbers forever)

```
def draw_br[illegible]ken_lin[illegible]ength):
  draw_[illegible]lin[illegible]h/3.)
  turtle.[illegible]
  draw_brok[illegible]ngth/3.)
  turtle.r[illegible]
  draw_[illegible]h/3.)
  turtle.[illegible](60)
  draw_broken_line(length/3.)
```

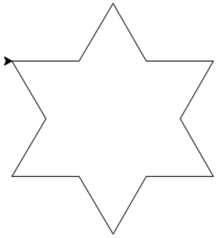
```
def draw_fractal_line(level, length):
  if level < 1:
    draw_normal_line(length)     # base case
  else:
    draw_fractal_line(level - 1, length/3.)
    turtle.left(60)
    draw_fractal_line(level - 1, length/3.)
    turtle.right(120)
    draw_fractal_line(level - 1, length/3.)
    turtle.left(60)
    draw_fractal_line(level - 1, length/3.)
```

Final Step of Koch Snowflake

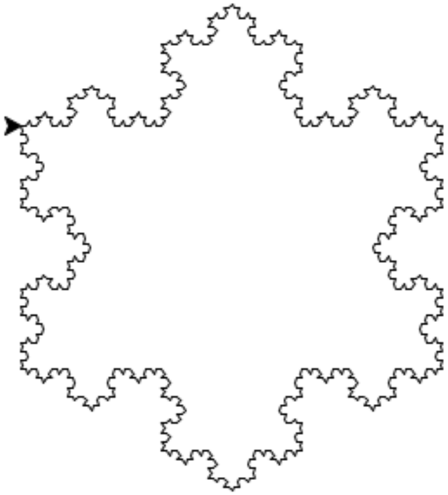
- The snowflake is three Koch broken lines done in a triangle



level = 0

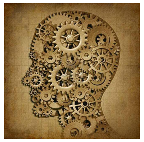


level = 1



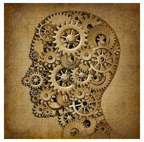
level = 4

- Using the broken line function write a function that draws a Koch snowflake



Pacing and Understanding

How well did you understand today?



- **A** Too easy, this lecture is way below my abilities
- **B** Everything went at a good pace, and I am fine
- **C** Too fast, but I will catch up on my own
- **D** Too fast, and I need you to slow down
- **E** I really do not think I can handle this