>>> f(1)
#
>>>
>>> f(2)
#
##
###
#
#
>>> f(3)
#
##
###
#
##
###
#
##
###
#
##
###
#
>>>
>>> f(4)  (TRY GUESSING!)
#
##
#
####
#
##
#
####
#
##
#
####
#
##
#
####
####
A recursive function

def f(n):
    if n == 1:
        print("#")
    else:
        f(n-1)
        print("#")
f(n-1)
A recursive function

def f(n):
    if n > 0:
        f(n-1)
        print("#"*n)
        f(n-1)
Recursive factorial

\[ n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 \]

But also:

\[ n! = n \times (n-1)! \]
\[ 0! = 1 \]

- “…” leads to a loop
- defining a function in terms of itself leads to recursion
Recursive factorial

def factorial(n):
    "returns n!, for any natural n>=0"
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
Defining in terms of itself?

Mandelbrot’s set

Emmy Noether (1892-1935)
Base and recursive cases

Every recursive function must have:

• One or more **base cases** (no more calls)
• One or more **recursive cases** (more calls)

Conditions:

• If “size” is $\leq 0$ we must be in a base case
• A recursive call must decrease “size” by at least 1
Base and recursive cases

Size decreases by 1 or more at each call
When size is \( \leq 0 \), we are in base case

This **guarantees termination** by the following property of natural numbers:

Every strictly decreasing sequence of natural numbers is finite

Not true for integers and real numbers
Base and recursive cases

\[ n! = (n+1)! / (n+1) \quad \text{True but not good} \]

\[ n! = n \times (n-1) \times (n-2)! \quad \text{Correct but watch out:} \]

def fact(n):
    if (n == 0):
        return 1
    else:
        return n*(n-1)*fact(n-2)
Computing integer powers

\[ \text{pow}(x,y) = x \times x \times x \times \ldots \times x \quad (y \text{ times}) \]

= \( x \times \text{pow}(x,y-1) \) \hspace{1cm} \text{(assume } x \text{ nonzero, } y>0; \ 0^0 \text{ undefined.)}

Base case?
Running time \( O(y) \)

Observation:
\[ x^{(2y)} = (x^2)^y \]

\[ \text{pow}(x,y) = \text{pow}(x\times x, y/2) \quad \text{if } y \text{ is even} \]
Slow powering

def pow(x, y):
    if y == 0:
        return 1
    else:
        return x * pow(x, y - 1)

y decreases by 1 at each call
running time O(y)
def pow(x, y):
    if y == 0:
        return 1
    elif y % 2 == 1:
        return x * pow(x, y-1)
    else:
        return pow(x*x, y/2)

y is divided by 2 every 2 calls
2 log \( y \) calls maximum, \( O(1) \) ops per call
running time \( O(\log y) \)

(alternatively, \( y \) in binary loses 1 bit at every \( /2 \))
Finding a zero of a continuous function

Given:
• a function \( f \) promised to be continuous
• a range \([a,b]\) such that \( f(a) < 0 < f(b)\)
• a margin \( \epsilon \)

Compute:
• a value \( x \) in \([a,b]\) such that \( f \) has a zero in \([x-\epsilon,x+\epsilon]\)

Such \( x \) exists by Bolzano’s theorem

def f(x):   return x*x - 2

>>> print(solver(f,0,4,0.000001))
1.4142136573791504

import math

>>> print(solver(math.sin,1,4,0.000001))
3.1415926218032837

>>> print(solver(math.cos,0,4,0.000001))
1.5707964897155762
Finding a zero of a continuous function

def solver(f, a, b, epsilon)

We keep the promise that $f(a)$ and $f(b)$ have different signs. So there must be a point in $[a,b]$ where $f(a) = 0$

Similar to binary search:
- Check the sign of $f((a+b)/2)$
- Discard half the interval

Termination:
- what decreases by at least 1 at each call?
Finding a zero of a continuous function

def solver(f, a, b, epsilon):
    c = (a+b)/2
    if abs(b-a) <= epsilon:
        return c
    else:
        if f(c) * f(a) < 0:
            return solver(f,a,c,epsilon)
        else:
            return solver(f,c,b,epsilon)

If we hit f(c) == 0, is there a problem?
Termination

Suppose that $|b-a|$ is approx $\epsilon \ast 2^k$

Then $|b-a|$ at the next call is

$$\epsilon \ast 2^k / 2 = \epsilon \ast 2^{(k-1)}$$

Number of calls = number of times we can divide ($\epsilon \ast 2^k$) by 2 before we get to $\epsilon$

... which is $k$

Isolating, we have $|b-a| = \epsilon \ast 2^k$ iff

$$k = \log_2 \left( \frac{|b-a|}{\epsilon} \right)$$
How is recursion executed?

Stack of calls made, with parameters

(example in blackboard)

Local variables local to the call
A new copy is created at each call

To note: recursive factorial uses memory $O(n)$ while iterative factorial uses memory $O(1)$
What is better, loops or recursion?

Every loop can be simulated with recursion
(next slides)

Recursion can be simulated with a loop and a stack
(blackboard; not trivial with multiple recursion)

Tail recursion can be replaced with a loop, with no stack
(next slides)
Loop to recursion

def f(x):
    y = a(x)
    return rec_f(x, y)

def f(x):
    y = a(x)
    while cond(x, y):
        (x, y) = b(x, y)
    return c(x, y)

def rec_f(x, y):
    if cond(x, y):
        return c(x, y)
    else:
        (x, y) = b(x, y)
        return rec_f(x, y)
def fact(n):
    f = 1
    while n > 0:
        f = f * n
        n = n - 1
    return f

def fact(n):
    f = 1
    return rec_fact(n,f)

def rec_fact(n,f):
    if n == 0:
        return f
    else:
        (f,n) = (f*n,n-1)
        return rec_fact(n,f)
def f(x):
    if cond(x):
        return c(x)
    else:
        z = b(x)
        return f(z)

def f(x):
    while cond(x):
        x = b(x)
    return c(x)

Tail recursion:
Nothing done after recursive call
Product of a list

\[ \text{prod}([v_1, v_2, v_3, \ldots, v_n]) = v_1 \times v_2 \times v_3 \times \ldots \times v_n \]

\[ = v_1 \times \text{prod}([v_2, \ldots, v_n]) \]

\[ = \text{prod}([v_1, \ldots, v_{n-1}]) \times v_n \]
Product of a list

def prod(lst):
    "returns the product of lst"
    if len(lst) == 0:
        return 1
    else:
        return lst[0] * prod(lst[1:])

Inefficiency: extracting & copying lst[1:]
def prod(lst):
    return rprod(lst, 0)

def rprod(lst, i):
    "returns the product of lst[i..len(lst)-1]"
    if (i == len(lst)):
        return 1
    else:
        return lst[i] * rprod(lst, i+1)
Product of a list

def prod(lst):
    return rprod(lst, len(lst)-1)

def rprod(lst, i):
    "returns the product of lst[0..i]"
    if (i == -1):
        return 1
    else:
        return rprod(lst, i-1) * lst[i]
Conclusions

• Loops and recursion have the same power in theory (if you can add a stack to loops)

• Often choice depends on elegance / naturality

• But some problems have natural multiple recursion solutions, complex iterative solutions

• Some languages automatically turn tail recursion to a loop

• Python DOES NOT optimize for tail recursion. You have to do the transformation by hand, if you want

• Remember that recursion may have “hidden memory usage”: stack of calls O(1) in loop may turn to O(n) in recursion

• So if tail recursive, in Python probably prefer loops

• Python has other ways (continuations, iterators, generators...)
Exercises

• Given a list of digits, compute the integer it represents  \([3,7,2,9]\) -> the int 3279

• Write a recursive version of binary search

• Apply the recursive->loop transformation to the recursive factorial. Note that you don’t get what you would expect

• The product-of-a-list functions are tail recursive. Turn them to loops