UC Berkeley EECS
Lecturer
Michael Ball

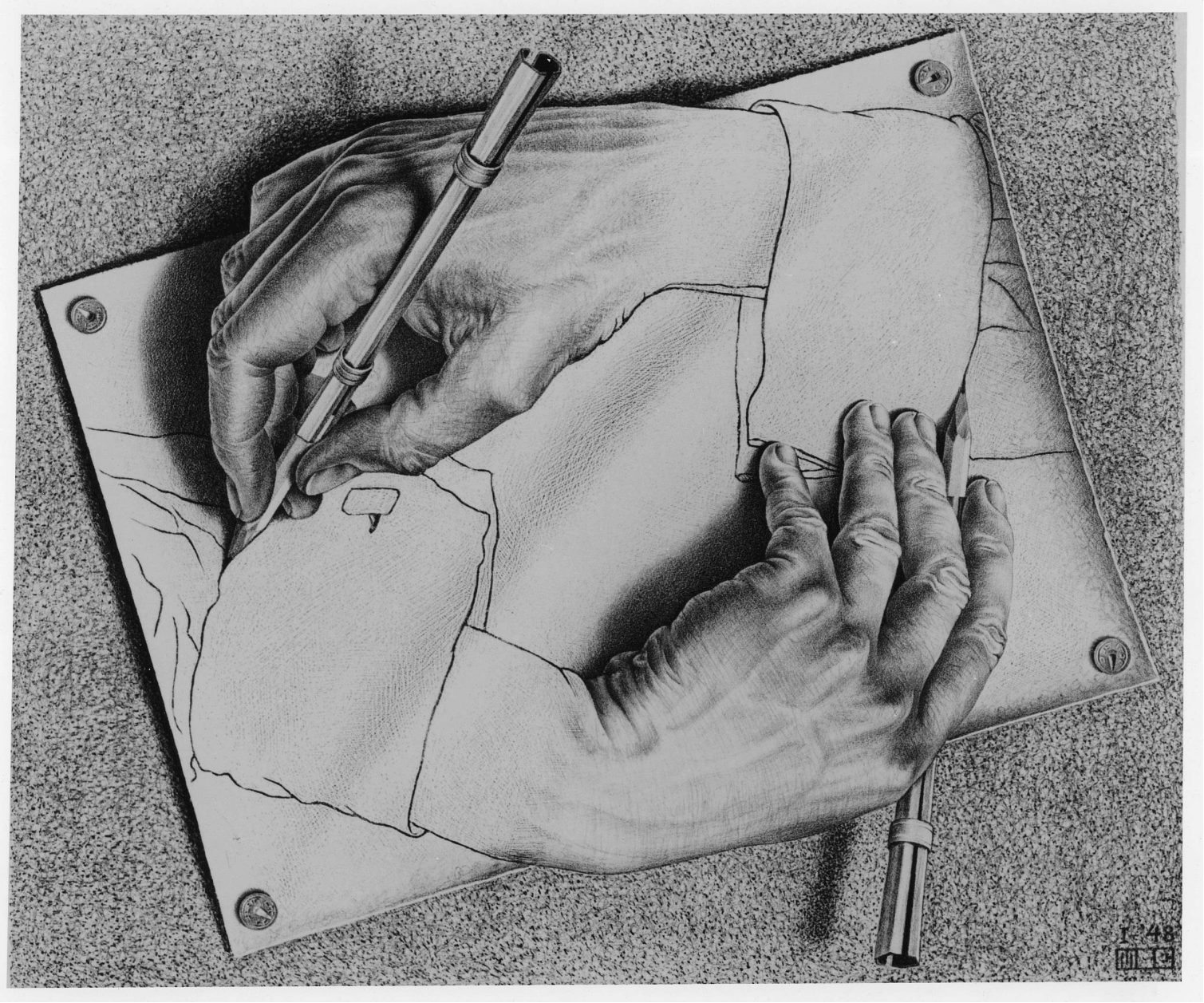
Computational Structures in Data Science

Lecture 5: Recursion

October 7, 2019

http://cs88.org

MC Escher “Drawing Hands” 1948



Administrative Issues

- **Midterm Next Week!**
- **7-9pm, Dwinelle 155**

Computational Concepts Toolbox

- Data type: values, literals, operations,
  - e.g., int, float, string
- Expressions, Call expression
- Variables
- Assignment Statement
- Sequences: tuple, list
  - indexing
- Data structures
- Tuple assignment
- Call Expressions
- Function Definition Statement
- Conditional Statement
- Iteration:
  - data-driven (list comprehension)
  - control-driven (for statement)
  - while statement
- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
- Recursion

Today: Recursion

re·cur·sion
/riˈkərZHən/

noun MATHEMATICS LINGUISTICS

the repeated application of a recursive procedure or definition.

- a recursive definition.
  plural noun: **recursions**

re·cur·sive
/riˈkərsiv/

adjective

characterized by recurrence or repetition, in particular.

- MATHEMATICS LINGUISTICS
  relating to or involving the repeated application of a rule, definition, or procedure to successive results.
- COMPUTING
  relating to or involving a program or routine of which a part requires the application of the whole, so that its explicit interpretation requires in general many successive executions.

- **Recursive function calls itself, directly or indirectly**

Why Recursion?

- "After Abstraction, Recursion is probably the 2nd biggest idea in this course"
- "It's tremendously useful when the problem is self-similar"
- "It's no more powerful than iteration, but often leads to more concise & better code"
- "It embodies the beauty and joy of computing"

Why Recursion? More Reasons

- **Recursive structures exist (sometimes hidden) in nature and therefore in data!**
- **It's mentally and sometimes computationally more efficient to process recursive structures using recursion.**

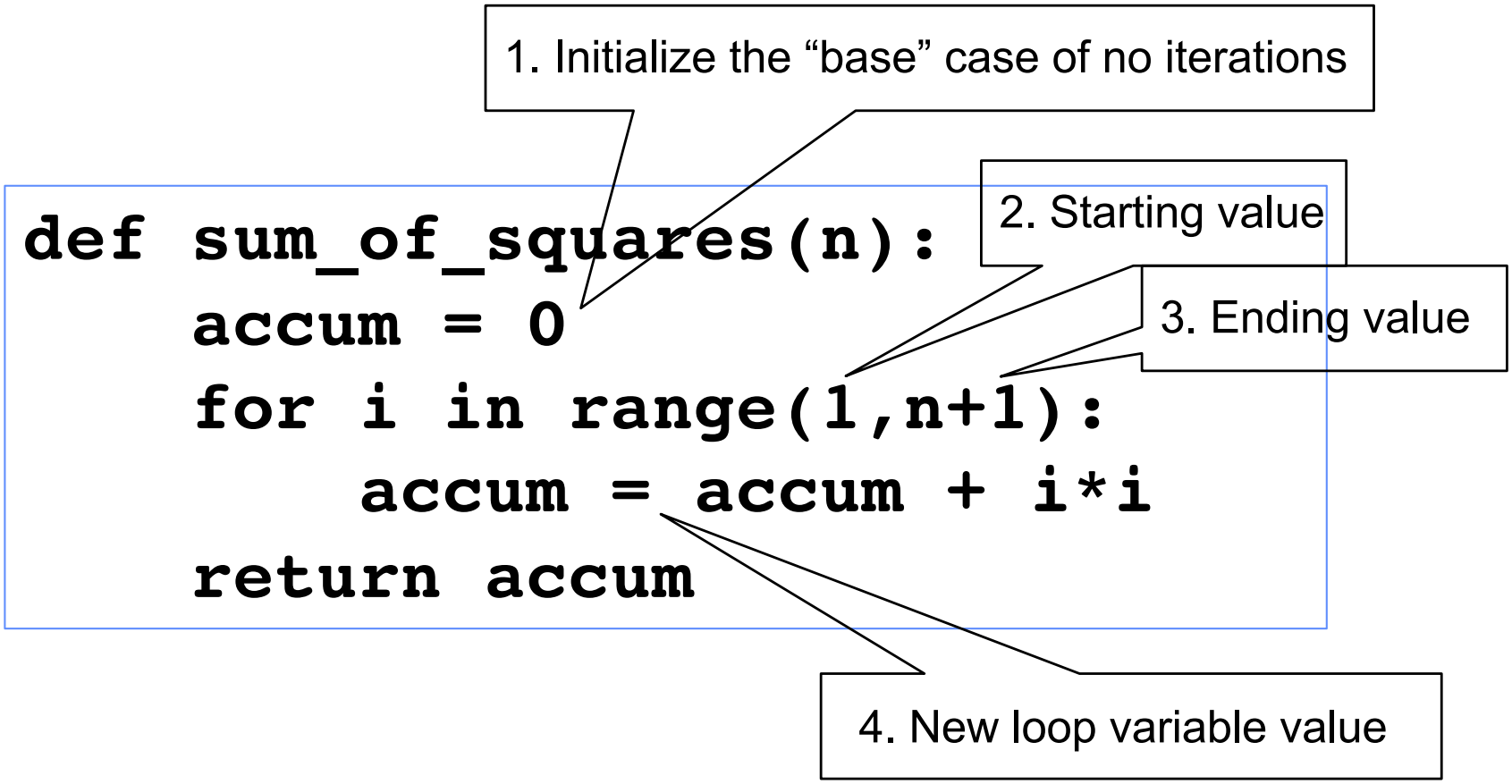
Function Review

- A function cannot…

  A) have a function as argument
  B) define a function within itself
  C) return a function
  D) call itself
  E) None of the above.

Solution:
E) A, B, C, D are all possible!

Recall: Iteration

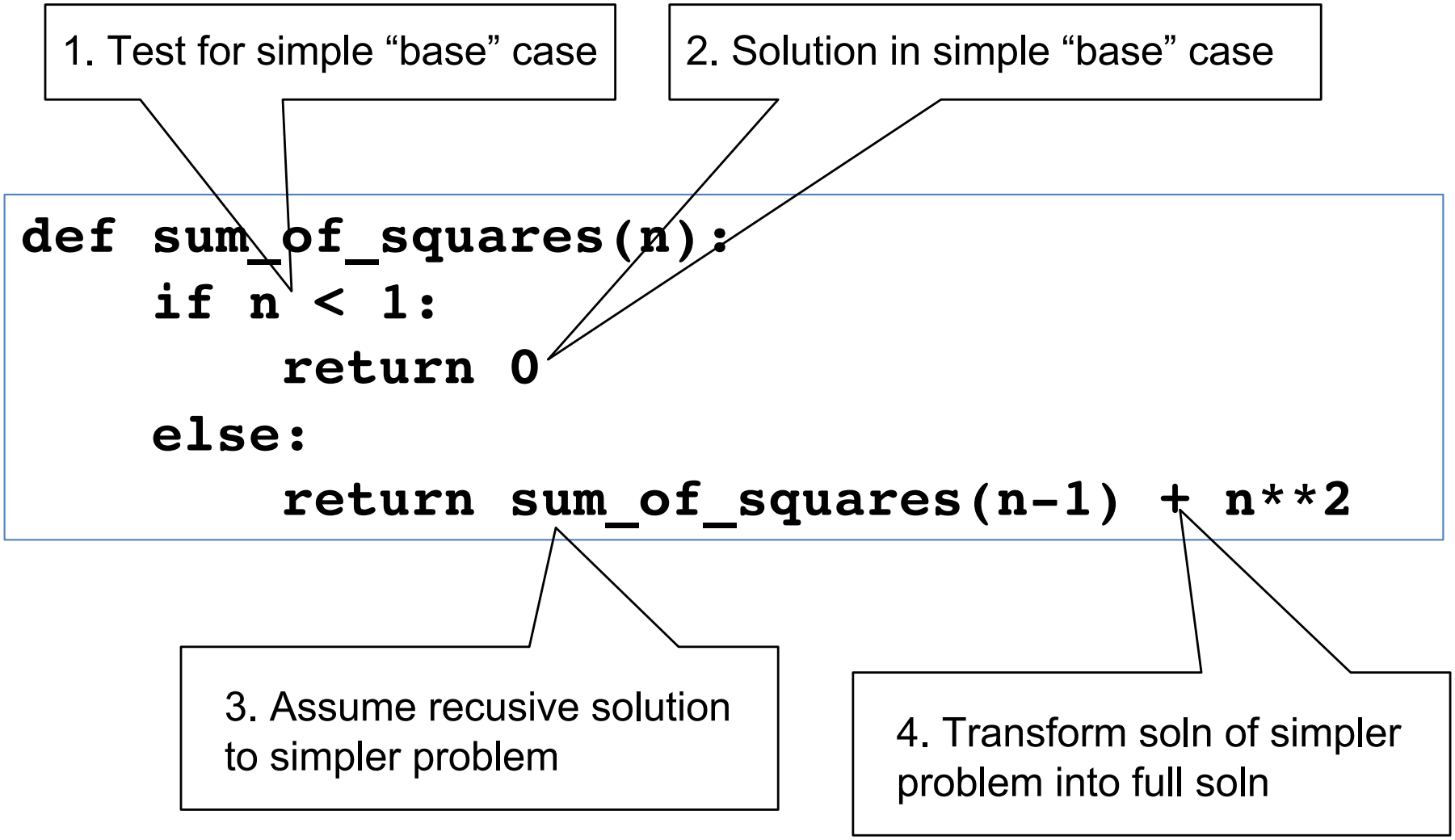



```
def sum_of_squares(n):
    accum = 0
    for i in range(1,n+1):
        accum = accum + i*i
    return accum
```

Demo Time

- **Vee a randomly recursive fractal**

Recursion Key concepts – by example



In words

- The sum of no numbers is zero
- The sum of 1^2 through n^2 is the
  - sum of 1^2 through $(n-1)^2$
  - plus n^2

```
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```

Why does it work

```
sum_of_squares(3)

# sum_of_squares(3) => sum_of_squares(2) + 3**2
#                   => sum_of_squares(1) + 2**2 + 3**2
#                   => sum_of_squares(0) + 1**2 + 2**2 + 3**2
#                   => 0 + 1**2 + 2**2 + 3**2 = 14
```

How does it work?

- Each recursive call gets its own local variables
  - Just like any other function call
- Computes its result (possibly using additional calls)
  - Just like any other function call
- Returns its result and returns control to its caller
  - Just like any other function call
- The function that is called happens to be itself
  - Called on a simpler problem
  - Eventually bottoms out on the simple base case

- Reason about correctness "by induction"
  - Solve a base case
  - Assuming a solution to a smaller problem, extend it

Questions

- In what order do we sum the squares ?
- How does this compare to iterative approach ?

```
def sum_of_squares(n):
    accum = 0
    for i in range(1,n+1):
        accum = accum + i*i
    return accum
```

```
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```

```
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return n**2 + sum_of_squares(n-1)
```

Tail Recursion

- All the work happens on the way down the recursion
- On the way back up, just return

```
def sum_up_squares(i, n, accum):
    """Sum the squares from i to n in incr. order"""
    if i > n:
        return accum
    else:
        return sum_up_squares(i+1, n, accum+i*i)

>>> sum_up_squares(1,3,0)
14
```

Base Case

Tail Recursive Case

Local variables

```
def sum_of_squares(n):
    n_squared = n**2
    if n < 1:
        return 0
    else:
        return n_squared + sum_of_squares(n-1)
```

- Each call has its own "frame" of local variables
- What about globals?

https://goo.gl/CiFaUJ

Iteration vs Recursion

For loop:

```
def sum(n):
    s=0
    for i in range(0,n+1):
        s=s+i
    return s
```

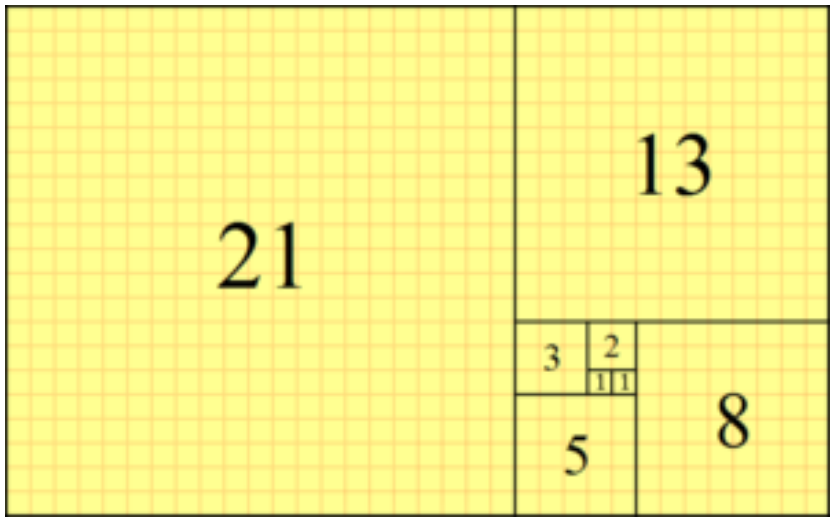
Iteration vs Recursion

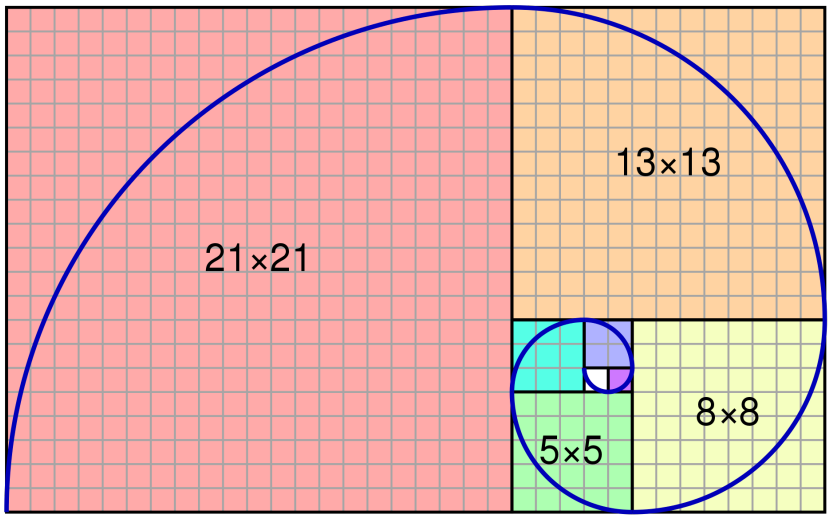
Recursion:

```
def sum(n):
    if n==0:
        return 0
    return n+sum(n-1)
```

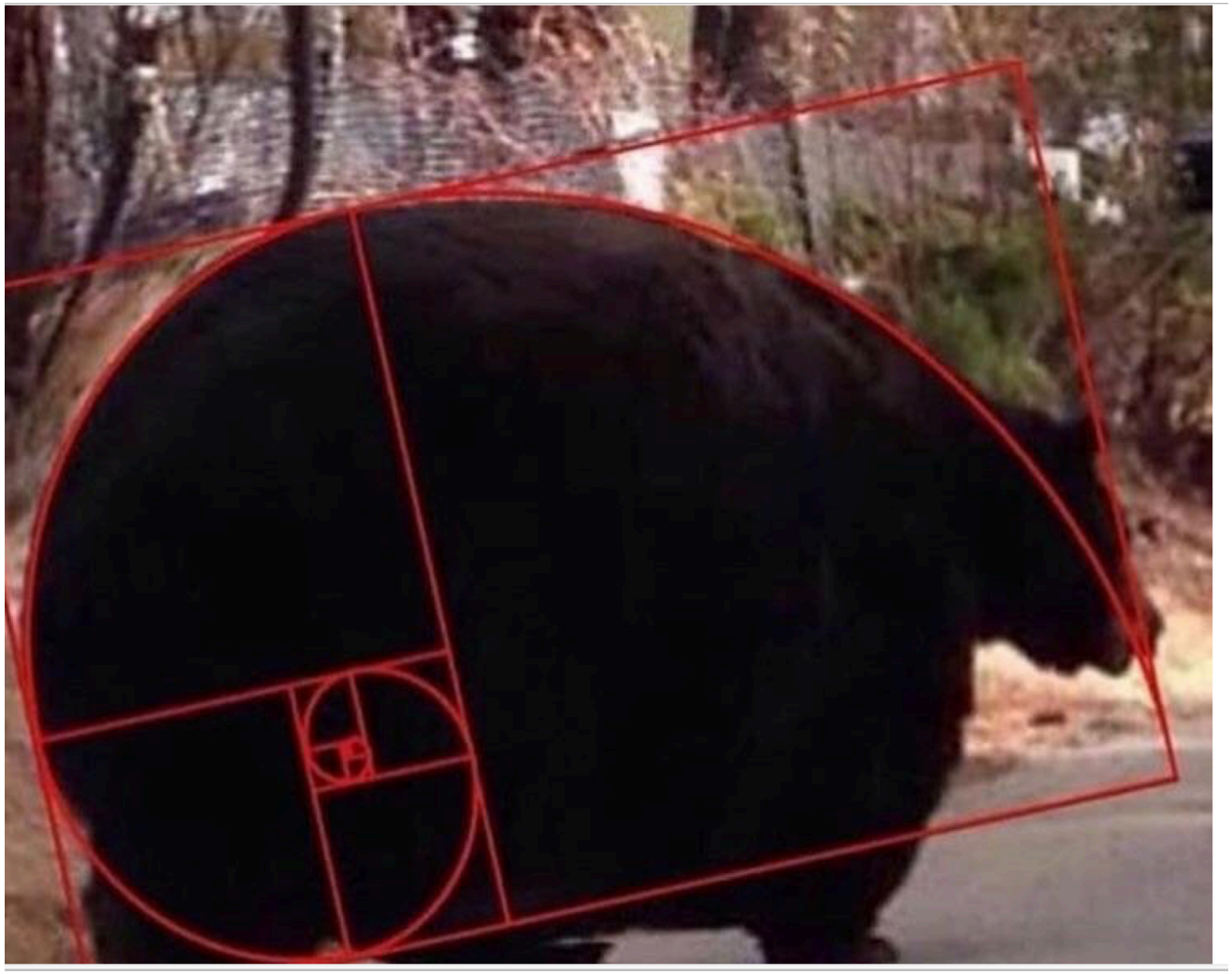
Fibonacci Sequence


```
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
       where fibonacci(1) == fibonacci(0) == 1
```

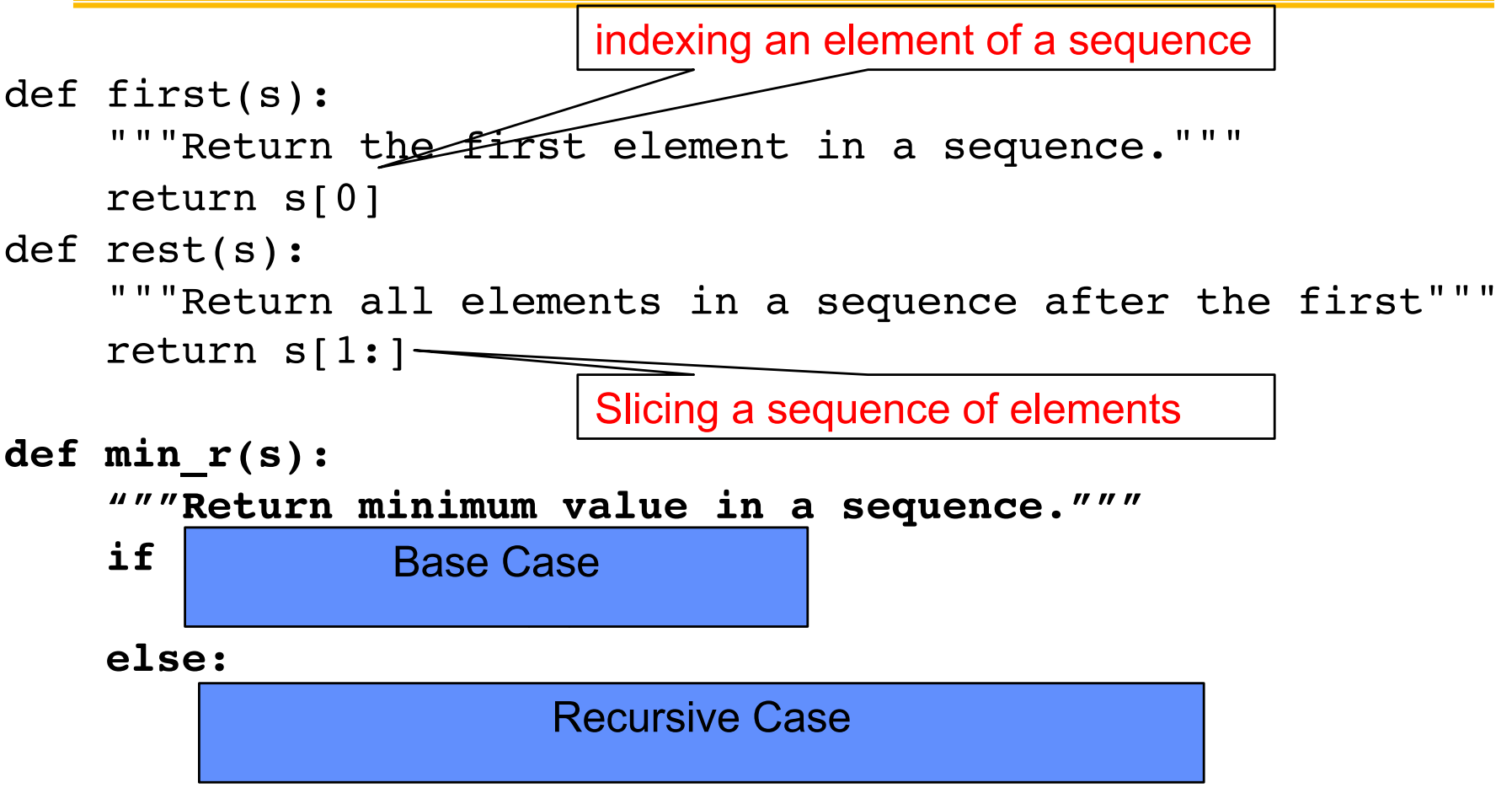




Go Bears!



Another Example



- **Recursion over sequence length, rather than number magnitude**

Visualize its behavior (print)

In [104]:
```
def min_r(s):
    print('min_r:', s)
    if len(s) == 1:
        return first(s)
    else:
        result = min(first(s), min_r(rest(s)))
        print('min_r:', s," => ", result)
        return result
```

In [105]:
```
min_r([3,4,2,5,11])
```

```
min_r: [3, 4, 2, 5, 11]
min_r: [4, 2, 5, 11]
min_r: [2, 5, 11]
min_r: [5, 11]
min_r: [11]
min_r: [5, 11]  =>  5
min_r: [2, 5, 11]  =>  2
min_r: [4, 2, 5, 11]  =>  2
min_r: [3, 4, 2, 5, 11]  =>  2
```

- **What about sum?**
- **Don't confuse print with return value**

Trust …

- **The recursive "leap of faith" works as long as we hit the base case eventually**

What happens if we don't?

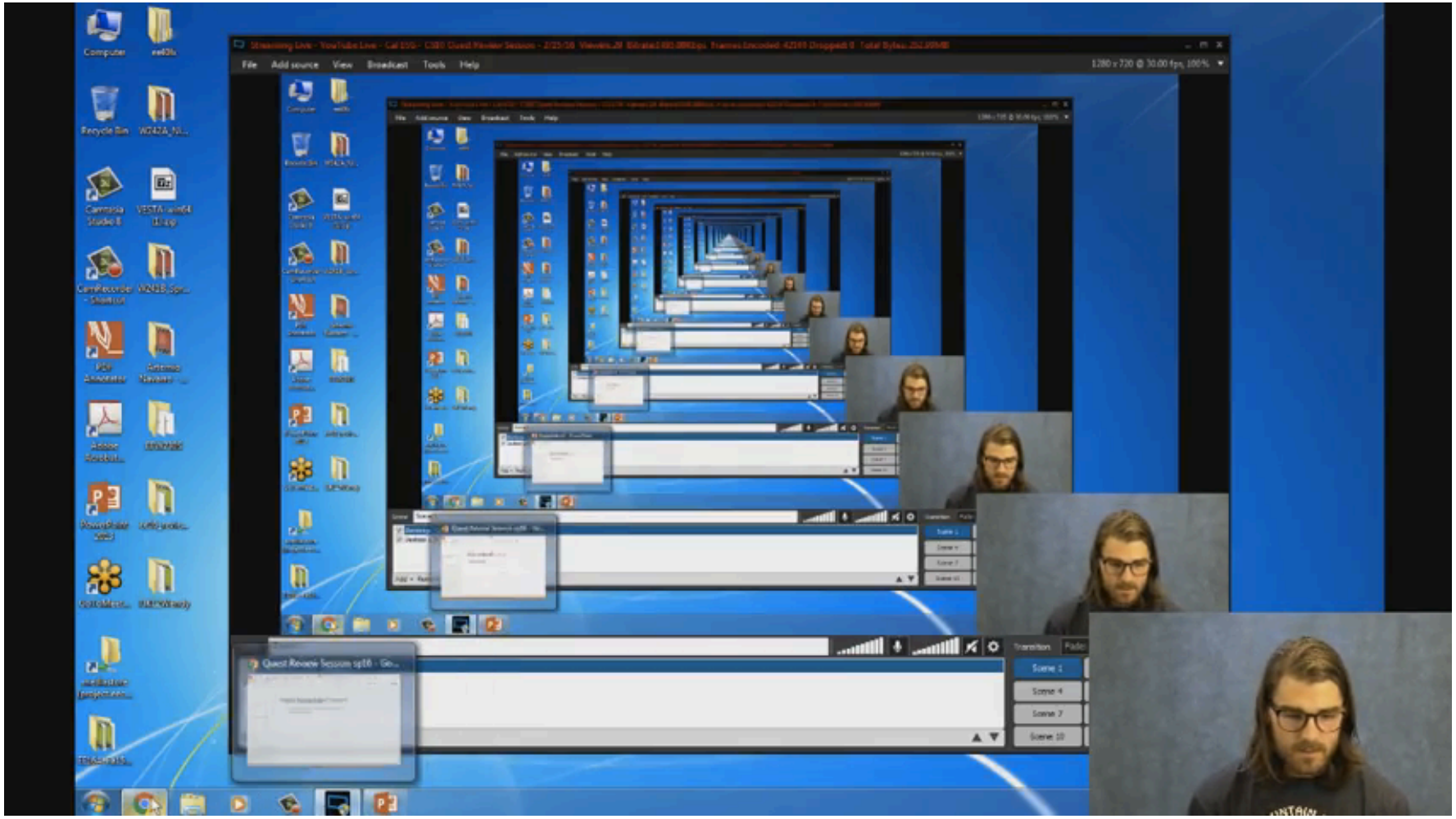
Recursion

- Recursion is…

A) Less powerful than a for loop
B) As powerful as a for loop
C) As powerful as a while loop
D) More powerful than a while loop
E) Just different all together

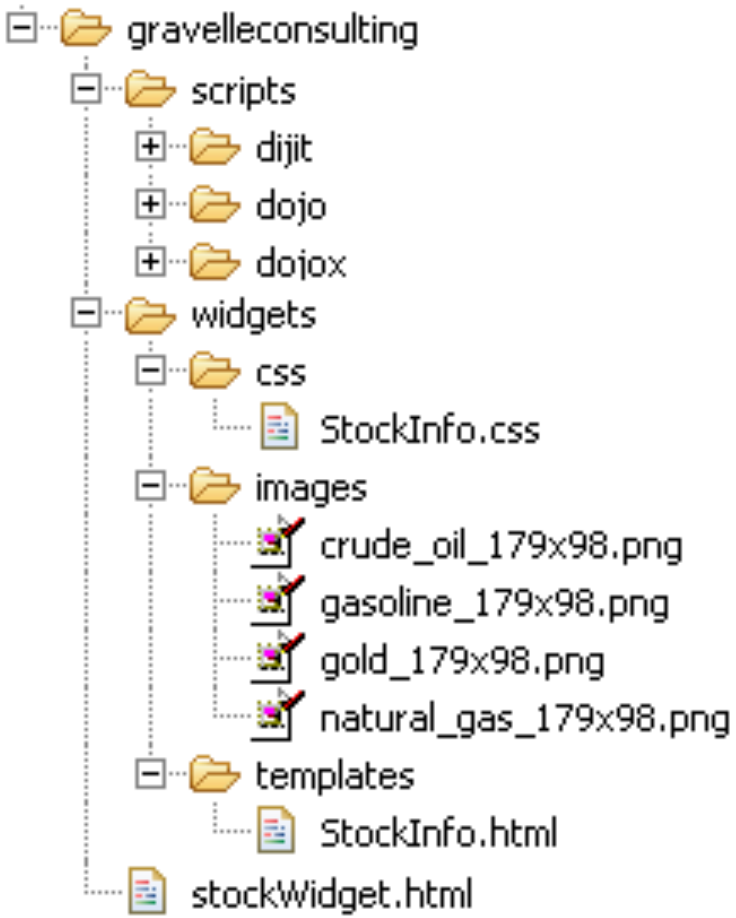
Solution:
C) Any recursion can be formulated as a while loop and any while loop can be formulated as a recursion (with a global variable).

Recursion (unwanted)



Example I

List all items on your hard disk



- **Files**
- **Folders contain**
  - **Files**
  - **Folders**

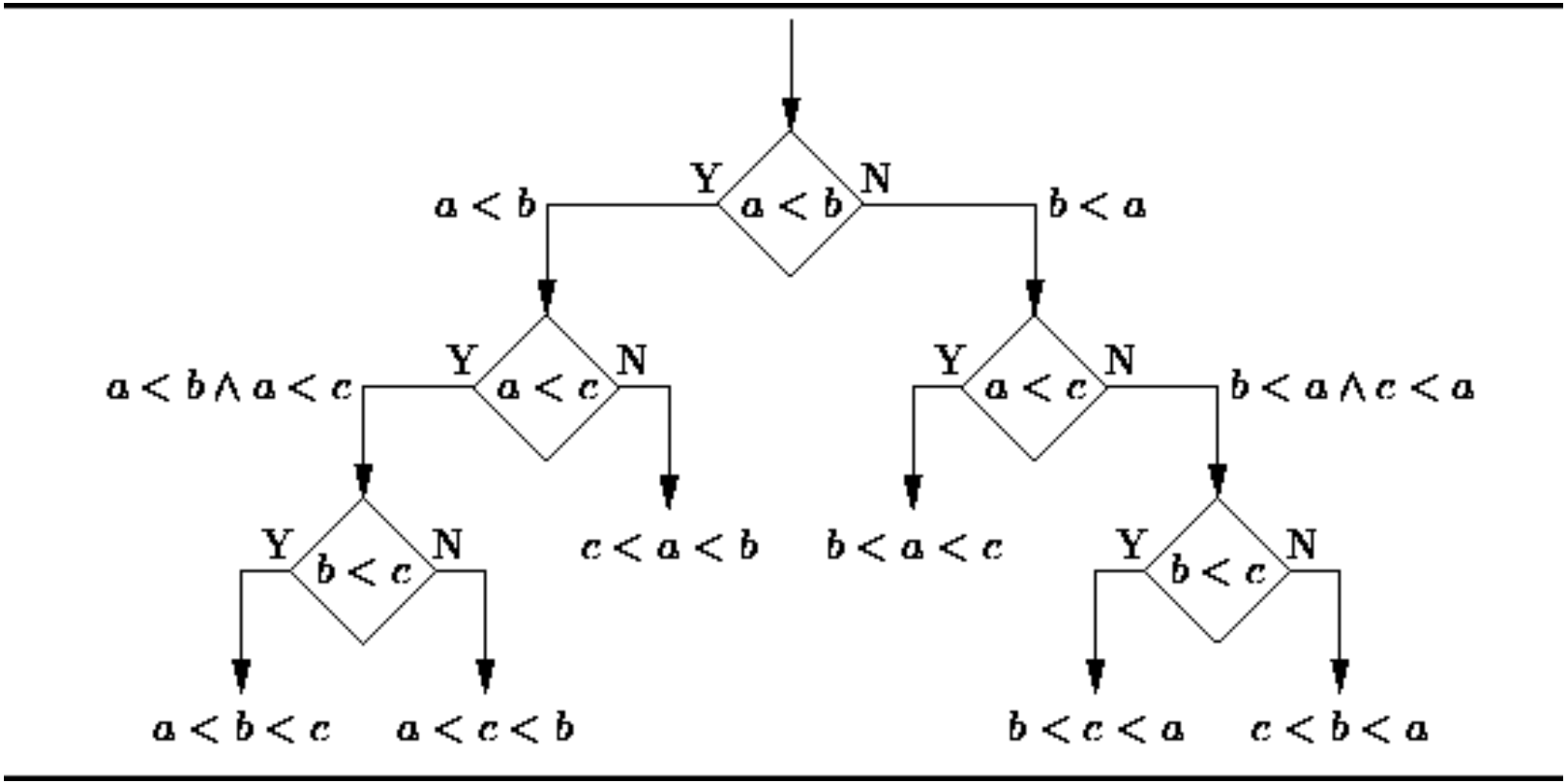
Recursion!

List Files in Python

```
def listfiles(directory):
    content = [os.path.join(directory, x) for x in os.listdir(directory)]

    dirs = sorted([x for x in content if os.path.isdir(x)])
    files = sorted([x for x in content if os.path.isfile(x)])

    for d in dirs:
        print d
        listfiles(d)

    for f in files:
        print f
```

Iterative version about twice as much code and much harder to think about.

Example II



Hidden recursive structure: Decision tree!