DECOMPOSITION, ABSTRACTION, FUNCTIONS, RECURSION

(download slides and .py files to follow along)

6.0001 LECTURE 4

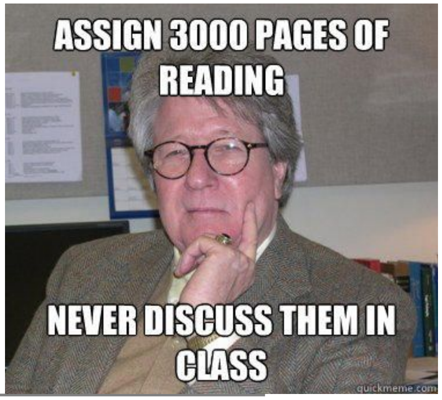
Eric Grimson

LAST TWO LECTURES

- while loops & for loops
  - should know how to write both kinds
  - should know when to use them
    - computations characterized by “state variables”; plus update rules for changing those variables on each iteration
    - *for* loops best when known range of iterations; *while* loops best when want to iterate until some condition is reached
- guess-and-check and approximation methods
  - trade off between accuracy and efficiency
- bisection method for fast algorithms when problem has an “ordering” property

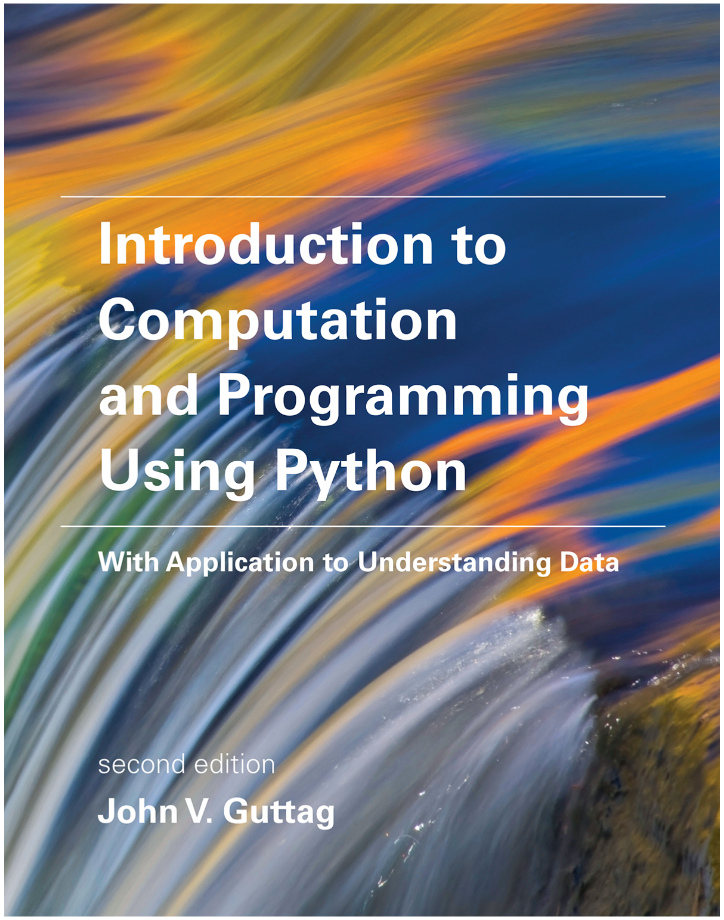
TODAY

- structuring programs and hiding details
- functions (aka procedures)
  - syntax & semantics
  - specifications
  - scope
- recursion



Assigned Reading

- today:
  - section 4.1 – 4.3
- next lecture:
  - section 5.1 – 5.5



See https://mitpress.mit.edu/books/introduction-computation-and-programming-using-python-second-edition for errata sheet

LEARNING TO PRODUCE CODE

- so far have covered basic language mechanisms
- in principle, you know all you need to know to accomplish anything that can be done by computation
  - after all, Turing showed that anything that is computable can be done with just 6 primitives!

- but in fact, we've taught you **nothing** about two of the most important concepts in programming…

DECOMPOSITION AND ABSTRACTION

- **decomposition:** how to divide a program into **self-contained parts** that can be combined to solve the current problem
  - ideally parts can be reused by other programs
  - self-contained means parts should complete computation using only inputs provided to them
- **abstraction:** how to ignore unnecessary detail
  - used to separate **what** something does, from **how** it actually does it
- the combination allows us to write complex code while suppressing details, so that we are not overwhelmed by the complexity

AN EXAMPLE: THE SMART PHONE

- a black box
  - can be viewed in terms of its inputs and outputs, and how outputs are related to inputs, without any knowledge of its internal workings
- user **doesn’t** know the details of how it works
- user **does** know the interface
- device converts a sequence of screen touches and sounds into expected useful functionality
- **abstraction:** We don’t need to know **how something works** to know **how to use it**

ABSTRACTION ENABLES DECOMPOSITION

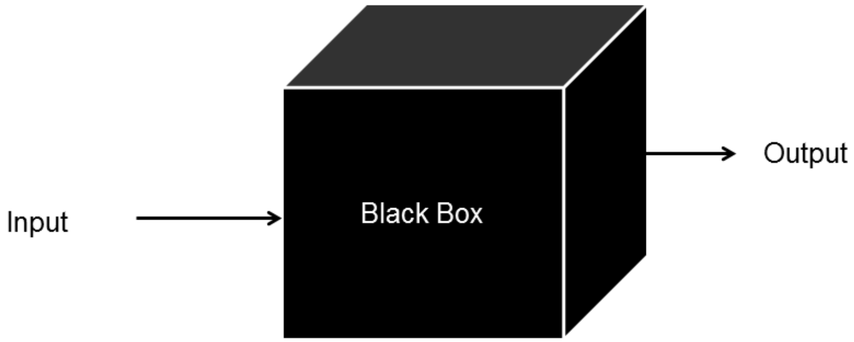
- 100's of distinct parts
- designed and made by different companies
  - do not communicate with each other
  - may use same subparts as others
- **decomposition:**

  Each component maker has to know **how its component interfaces** to other components, but **not how other components are implemented**; can solve sub-problems independently

True for hardware **and** for software

OUR GOAL

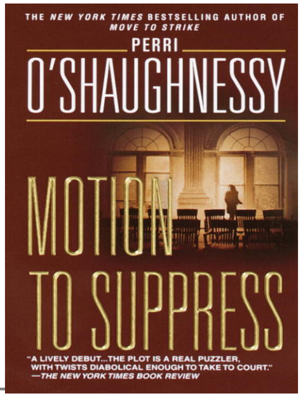
Apply these concepts of abstraction (black box) and decomposition (splitting into self-contained, possibly nested parts) to programming!



Internal behavior of the code is unknown

Many black boxes, can be used together without knowing details of interiors

SUPPRESS DETAILS with ABSTRACTION



- in programming, think of a piece of code as a **black box**
  - user **cannot** see details (in fact, want to hide tedious coding details)
  - user does not **need** to see details
  - user does not **want** to see details
  - coder creates details, and designs interface

- achieve abstraction with **function (or procedure)**
  - **function** lets us capture code within a black box
  - function has **specifications**, captured using **docstrings**
  - think of **docstring** as "contract" between creator and user:
    - if user provides **input** that satisfies stated conditions, function will produce **output** according to specs, with indicated **side effects**
    - not typically enforced in Python (we'll see assertions later), but user relies on coder's work meeting the contract

CREATE STRUCTURE with DECOMPOSITION

- in programming, divide code into **modules** that are:
  - **self-contained** (can compute using basic elements and inputs provided to them)
  - used to **break up** code into logical pieces
  - intended to be **reusable**
  - used to keep code **organized**
  - used to keep code **coherent** (readable and understandable)
- in this lecture, achieve decomposition with **functions**
- in a few lectures, achieve decomposition with **classes**
- decomposition relies on abstraction to enable construction of complex modules from simpler ones

ABSTRACTION'S VIRTUOUS CYCLE

- start with primitives (e.g., 4, 3, +, *)
- have ways to combine into more complex expressions (e.g., (4+3)*8 + 3**(8-3))
- about to add ways to capture complex expressions

```
def crazy(a, b, c):
    return (a+b)*c + b**(c-b)
```

We will see how this captures a process in a function shortly

- now can treat function `crazy` as if it is a built-in primitive
- repeat cycle

FUNCTIONS

com·put·er ex·pert
[kəmˈpyoodər ˈekˌspərt] *noun*
someone who has not read the instructions, but who will nevertheless feel qualified to install a program and, when it does not function correctly, pronounce it incompatible with the operating system

- write reusable pieces of code, called **functions** or **procedures**
- functions are not run until they are "**called**" or "**invoked**" in a program
  - compare to code in a file that runs as soon as you load it
- function characteristics:
  - has a **name** (there is an exception we won't worry about for now)
  - has (formal) **parameters** (0 or more) — Names for input values
  - has a **docstring** (optional but recommended) — Describes behavior
    - a comment delineated by """ (triple quotes) that provides a **specification** for the function – contract relating output to input
  - has a **body** — Instructions to evaluate using inputs
  - **returns** something (typically) — Output given back to invoker

HOW TO WRITE & CALL (INVOKE) A FUNCTION

keyword

name

parameters or arguments

May have 0, 1 or more parameters
Separated by commas

some special strings reserved, cannot use as name of function

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0

is_even(3)
```

indentation defines extent of function body

specification, docstring

body

later in the code, you call (or invoke) the function using its name and providing values for parameters

IN THE FUNCTION BODY

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0
```

run some commands

keyword

expression to evaluate and return to invoker

- if function invoked in shell, value returned to shell; in which case value printed
- if function invoked within other computation, value return to invoker

ENVIRONMENTS

- global environment is place where user interacts with Python interpreter
  - contains bindings of variables to values from loading files, from user interaction with interpreter, and Python built-ins
- invoking a function creates a new environment (frame)
  - formal parameters bound to values passed to function
  - body of function evaluated with respect to this frame
    - any reference to a parameter uses value associated with parameter binding
    - frame inherits bindings from frame in which function called; thus references to variables other than formal parameters get values through this inheritance

VARIABLE SCOPE

- new **scope/frame/environment** created when function is called
- **formal parameter** gets bound to the value of **actual input parameter** when function is called
- **scope** is mapping of names to objects; defines context in which body is evaluated – values of variables given by bindings of names

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

y = 3
z = f( y )
```

formal parameter

Function definition

actual parameter

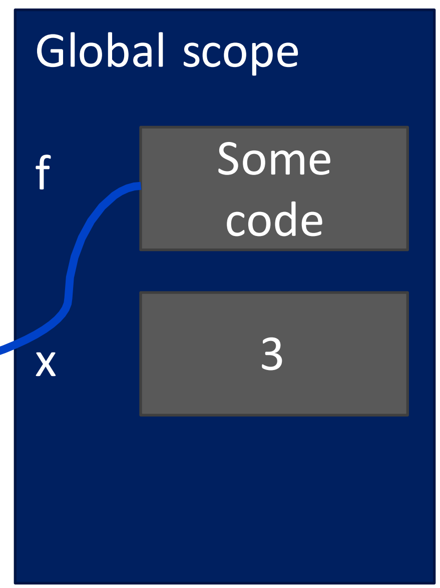
Main program code
* initializes a variable x
* makes a function call f(x)
* assigns return of function to variable z

Can be any legal value

VARIABLE SCOPE

After evaluating `def` and executing 1st assignment

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



NOTE: this code is not yet evaluated; simply exists as text

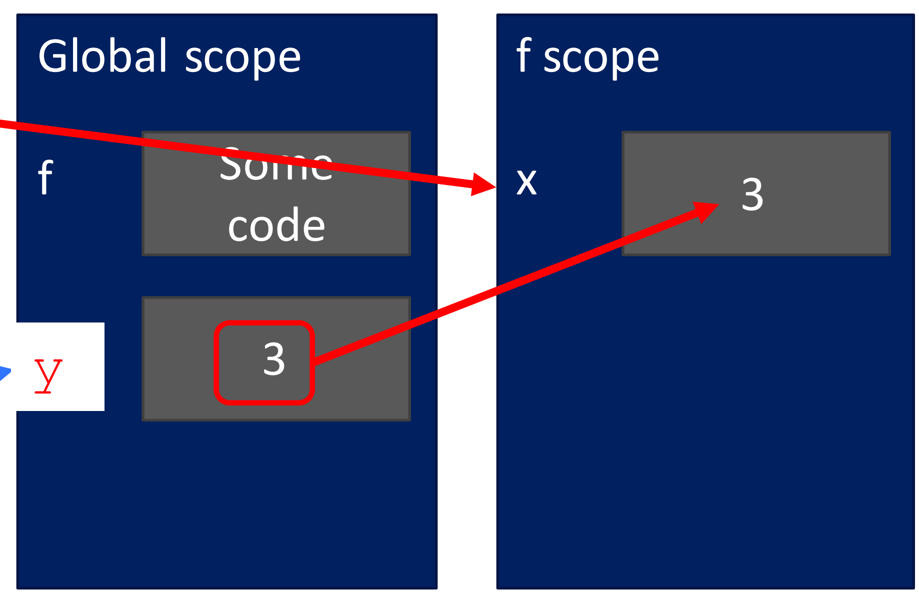
VARIABLE SCOPE

After f invoked

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x
```

Because we are evaluating this expression in interpreter

```
y = 3
z = f( y )
```



VARIABLE SCOPE

Evaluating body of f

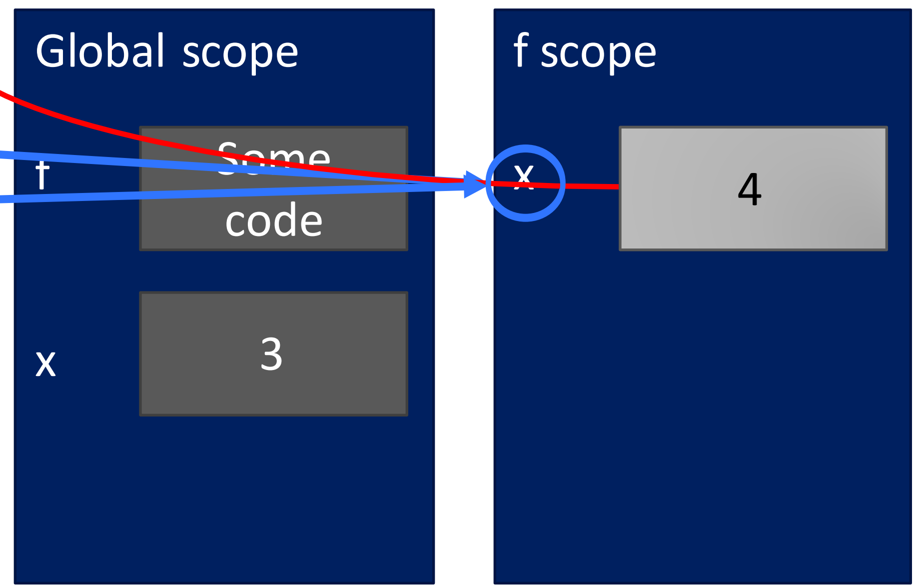
Note where binding for x is changed: in frame created by invocation of f, since body evaluated with respect to this frame

`in f(x): x = 4` printed out

state just before return

Because we are evaluating this expression in scope of f

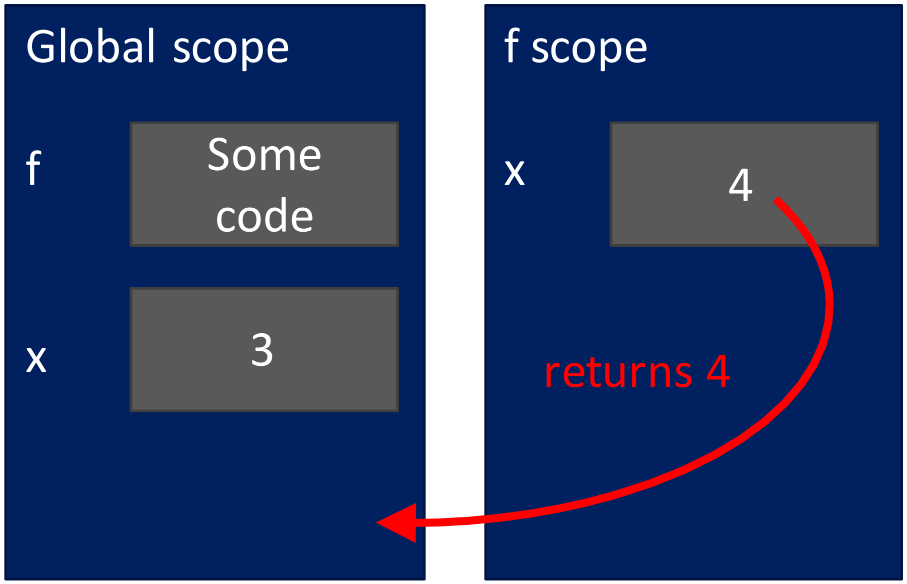
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



VARIABLE SCOPE

During the return

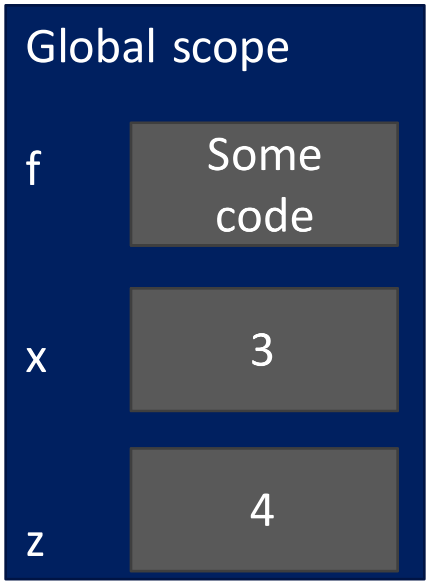
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



VARIABLE SCOPE

After executing 2nd assignment

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



WHAT IF THERE IS NO `return`


```
def is_even( i ):
    """
    Input: i, a positive int
    Does not return anything
    """
    i%2 == 0
```

without a return statement

- Python returns the value **None, if no `return` given**
- represents the absence of a value
  - if invoked in shell, nothing is printed
- no static semantic error generated

YOUR TURN


```
def add(x,y):
    return x+y
def mult(x,y):
    print(x*y)

add(1,2)
print(add(2,3))
mult(3,4)
print(mult(4,5))
```

What is printed in the console if your run this code as a file?

A) Nothing

B) 5
   12
   20
   None

C) 3
   5
   12
   20

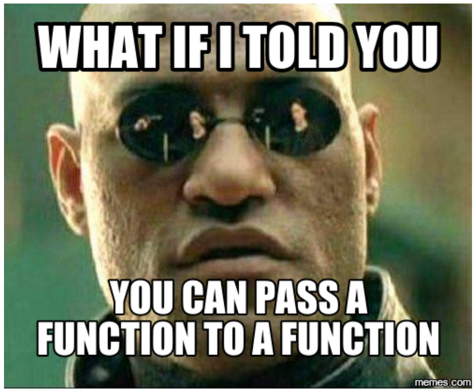
D) 5
   20

`return` vs. `print`

- return only has meaning **inside** a function
- only **one** return executed inside a function
- code inside function but after return statement not executed
- has a value associated with it, **given to function caller**

- print can be used **outside** functions
- can execute **many** print statements inside a function
- code inside function can be executed after a print statement
- has a value associated with it, **outputted** to the console
- print expression itself returns `None` value

FUNCTIONS AS PARAMETERS



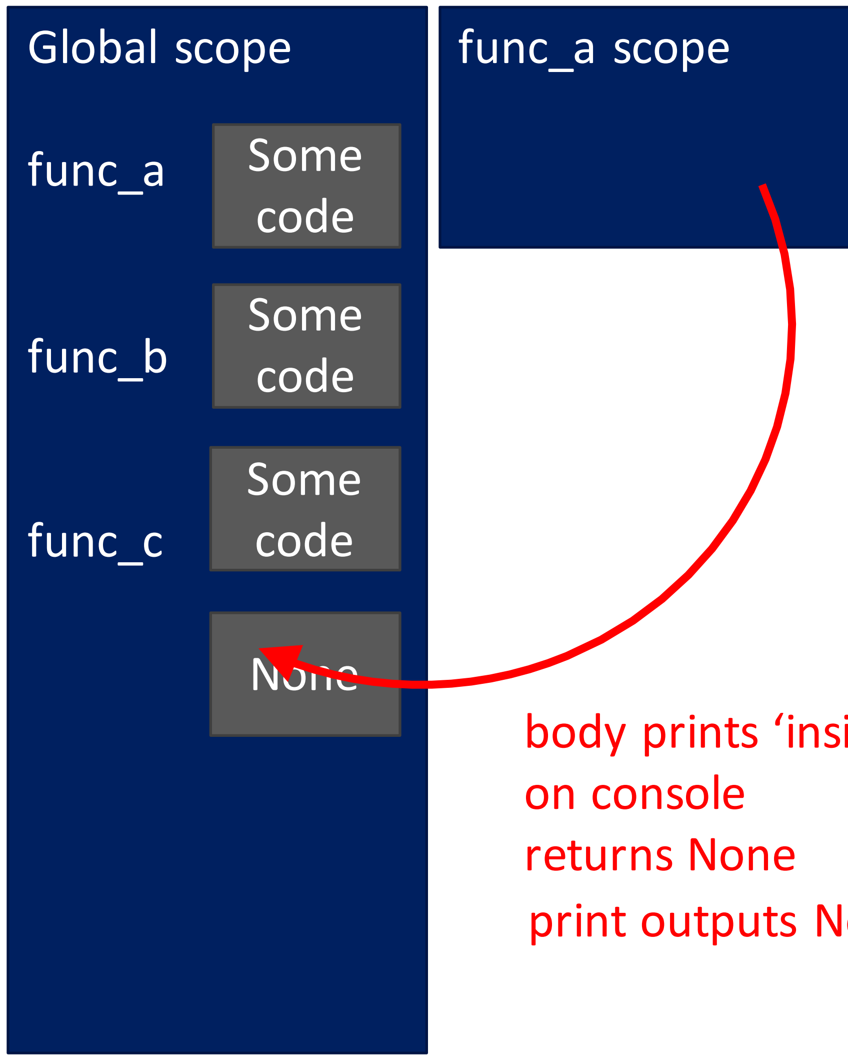
- parameters can take on any type, even functions

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```

call func_a, takes no parameters
call func_b, takes one parameter, an int
call func_c, takes two parameters, another function and an int

FUNCTIONS AS PARAMETERS

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



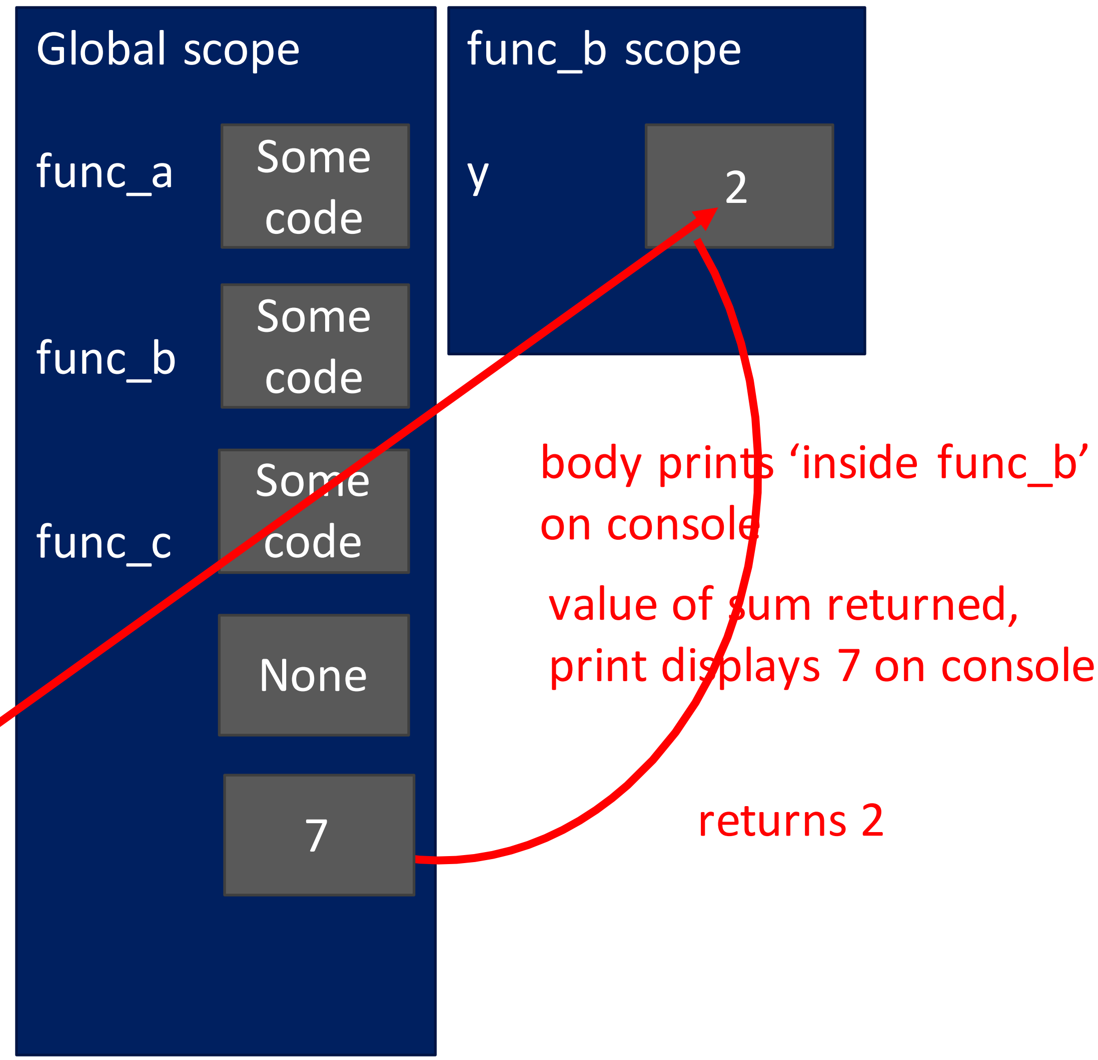
No bindings, as no parameters

But note form of invocation

body prints 'inside func_a' on console
returns None
print outputs None

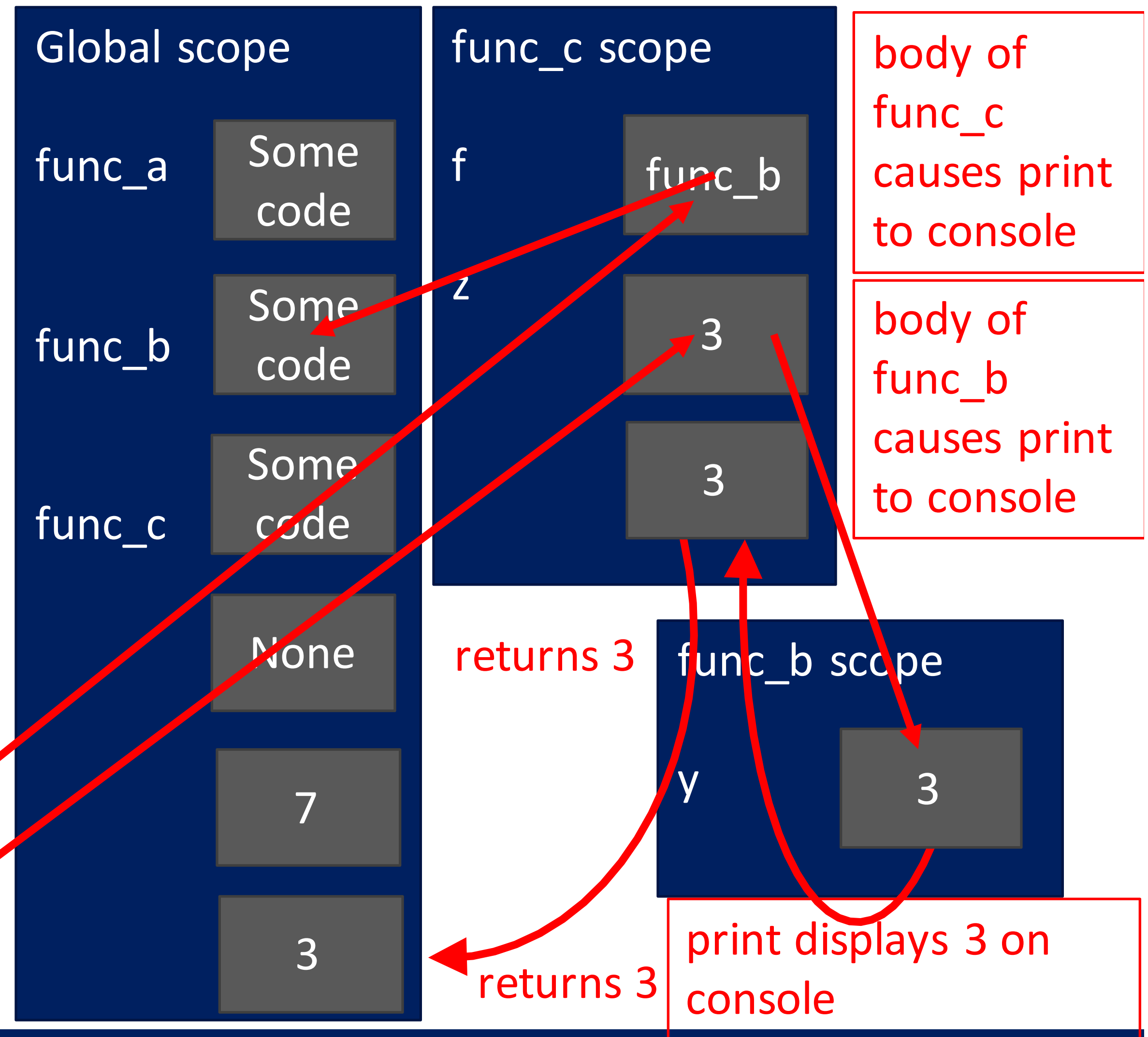
FUNCTIONS AS PARAMETERS

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



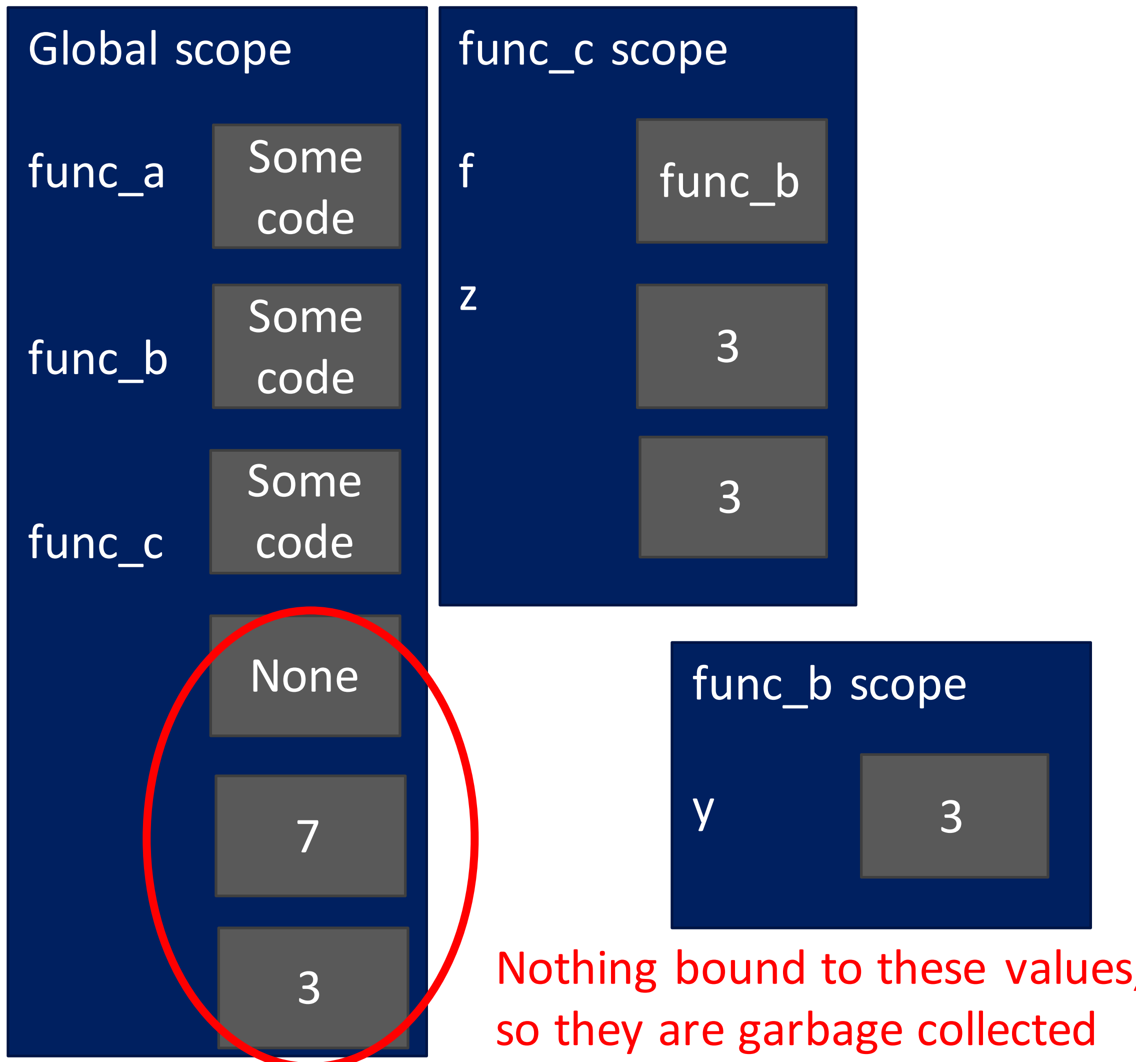
FUNCTIONS AS PARAMETERS

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



FUNCTIONS AS PARAMETERS

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



YOUR TURN


```
def sq(func,x):
    y = x**2
    return func(y)
def f(x):
    return x**2
calc = sq(f,2)
print(calc)
```

What does this code print?

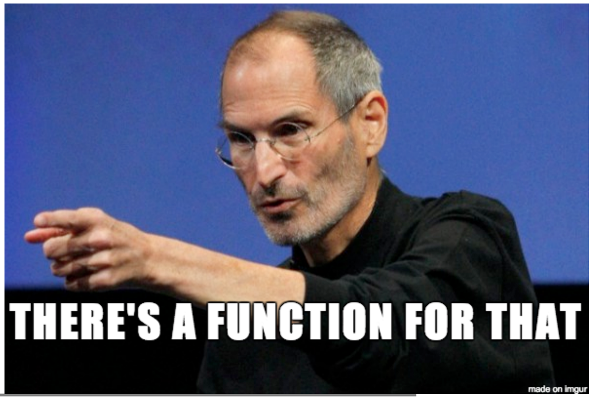
A) 4

B) 8

C) 16

D) nothing, it will show an error

FUNCTIONS CAN RETURN FUNCTIONS



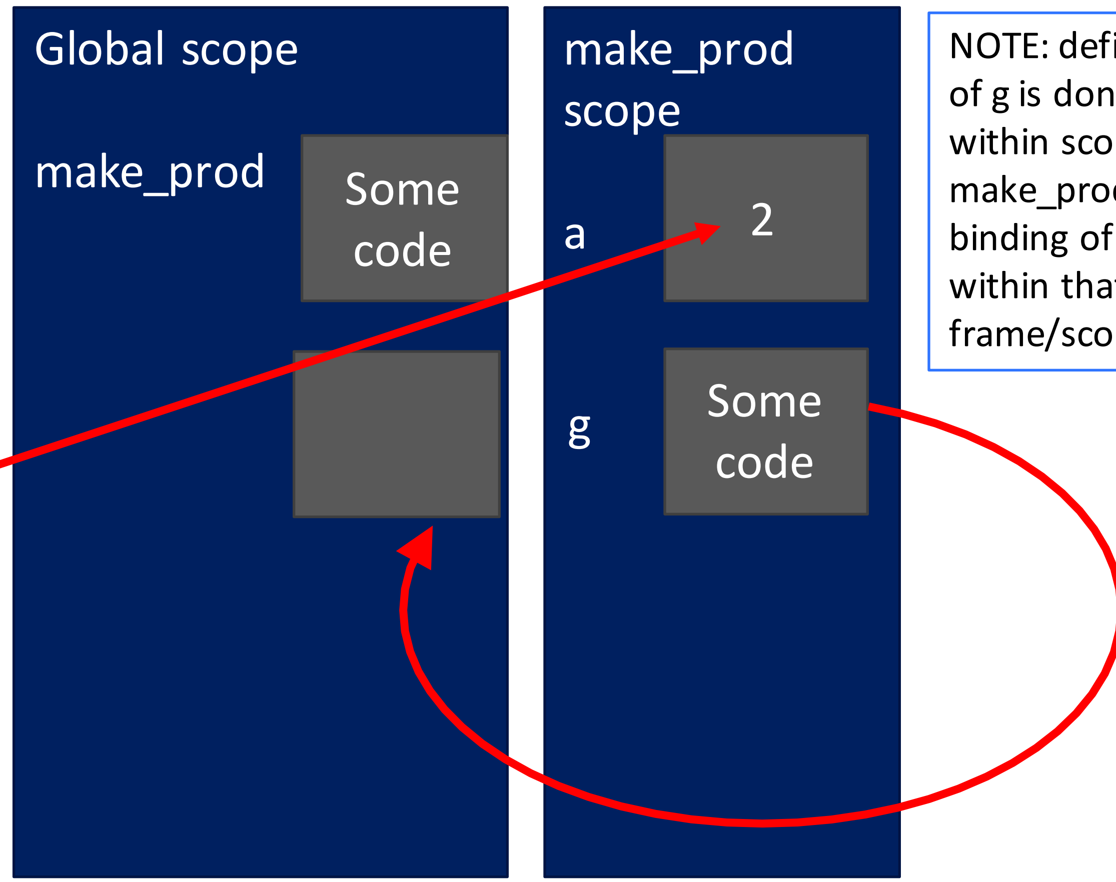

```
def make_prod(a):
    def g(b):
        return a*b
    return g


val = make_prod(2)(3)
print(val)
```

OR

```
doubler = make_prod(2)
val = doubler(3)
print(val)
```

SCOPE DETAILS

```
def make_prod(a):
    def g(b):
        return a*b
    return g


val = make_prod(2)(3)
print(val)
```

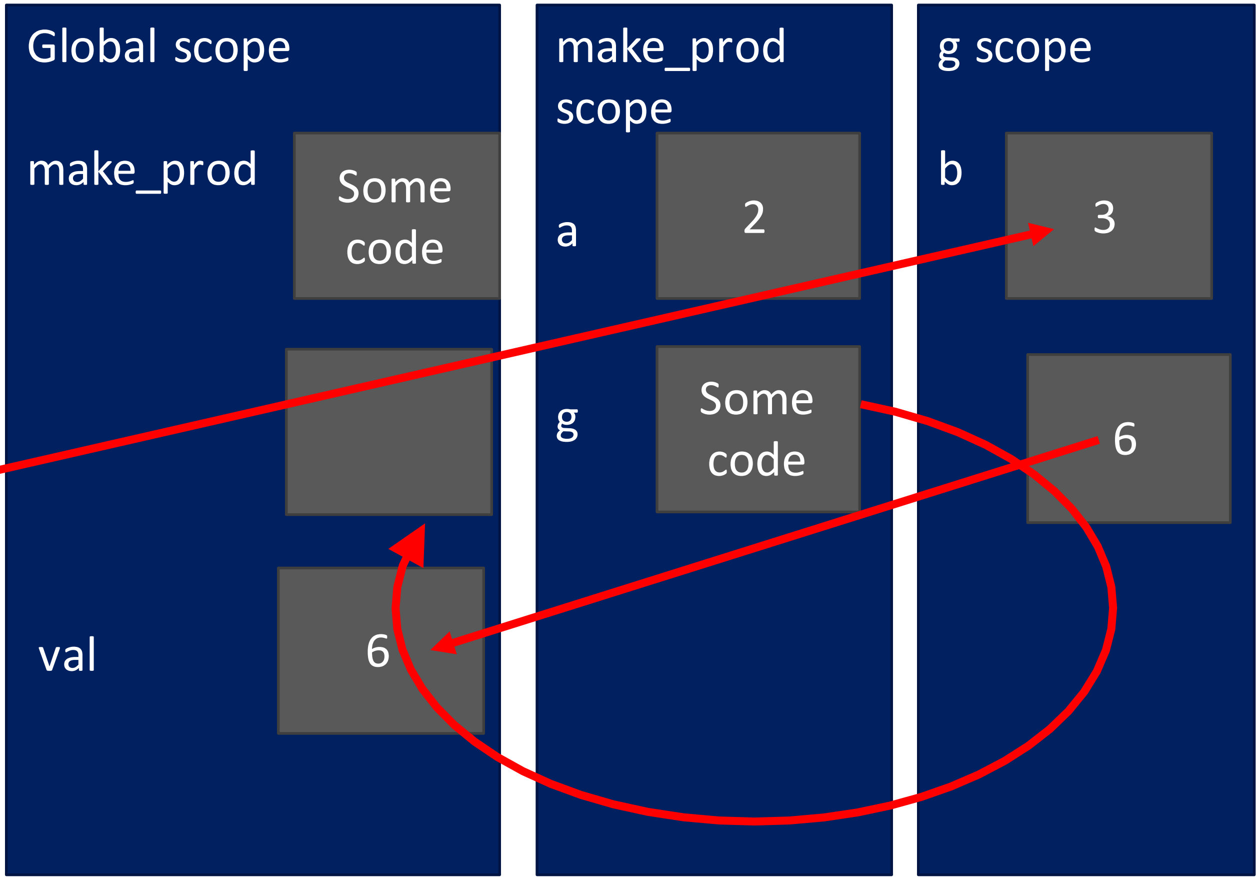


NOTE: definition of g is done within scope of make_prod, so binding of g is within that frame/scope

Returns pointer to g

SCOPE DETAILS

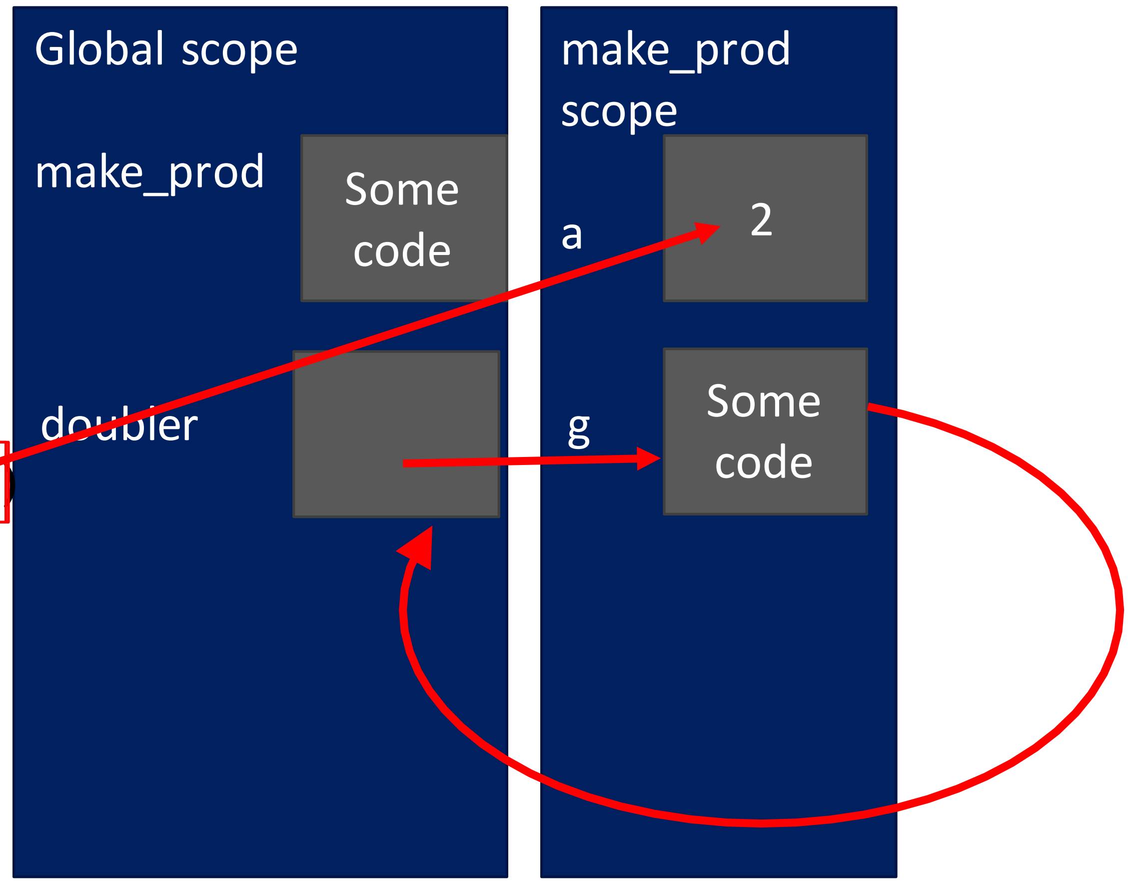
```
def make_prod(a):
    def g(b):
        return a*b
    return g

val = make_prod(2)(3)
print(val)
```



code can see both b and a values

SCOPE DETAILS

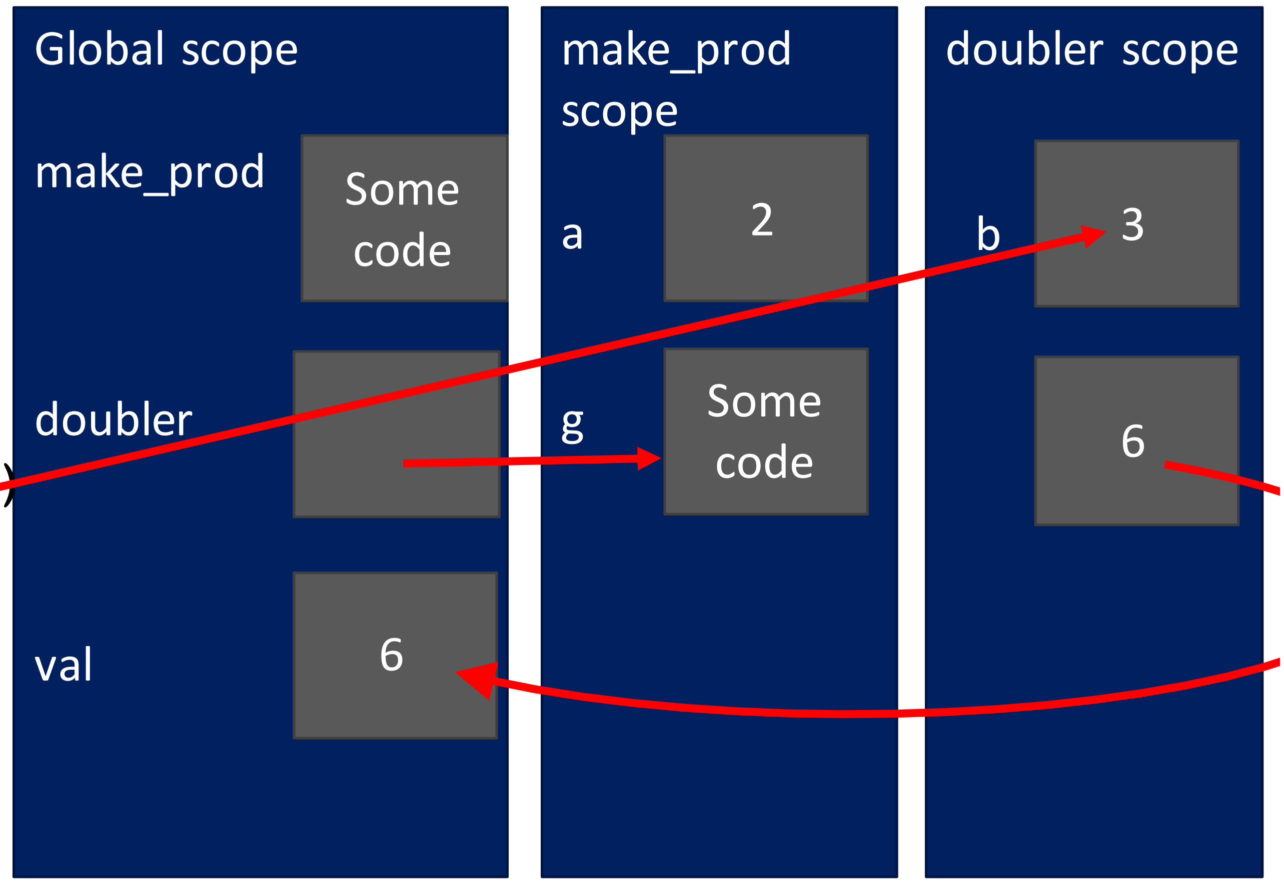
```
def make_prod(a):
    def g(b):
        return a*b
    return g

doubler = make_prod(2)
val = doubler(3)
print(val)
```



Returns pointer to g

SCOPE DETAILS

```
def make_prod(a):
    def g(b):
        return a*b
    return g

doubler = make_prod(2)
val = doubler(3)
print(val)
```



doubler code can see both b and a values

Returns value

SCOPE EXAMPLE

- inside a function, **can access** a variable defined outside
- inside a function, **cannot modify** a variable defined outside -- can using **global variables**, but frowned upon

```
def f(y):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

x is re-defined in scope of f

different x objects

```
2
5
```

```
def g(y):
    print(x)
    print(x + 1)

x = 5
g(x)
print(x)
```

x from outside g

x inside g is picked up from scope that called function g

```
5
6
5
```

```
def h(y):
    x += 1

x = 5
h(x)
print(x)
```

UnboundLocalError: local variable 'x' referenced before assignment

```
Error
```

SCOPE EXAMPLE

- inside a function, **can access** a variable defined outside
- inside a function, **cannot modify** a variable defined outside -- can using **global variables**, but frowned upon

```
def f(y):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

```
def g(y):
    print(x)


x = 5
g(x)
print(x)
```

```
def h(y):
    x += 1

x = 5
h(x)
print(x)
```

x from global/main program scope

HARDER SCOPE EXAMPLE

IMPORTANT
and
TRICKY!

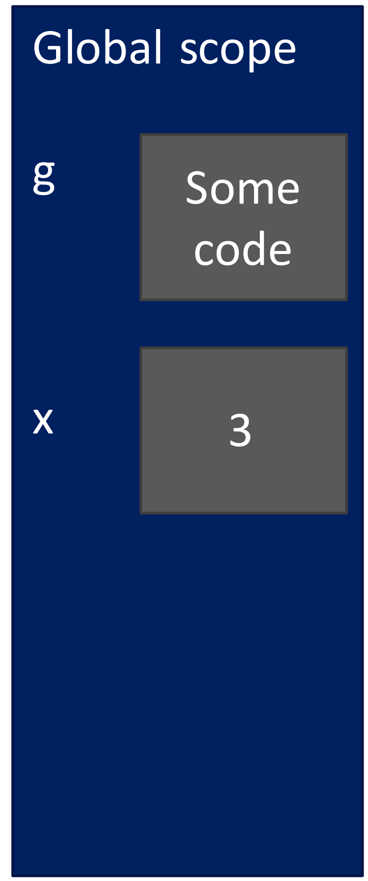
Python Tutor is your best friend to help sort this out!

http://www.pythontutor.com/

SCOPE DETAILS

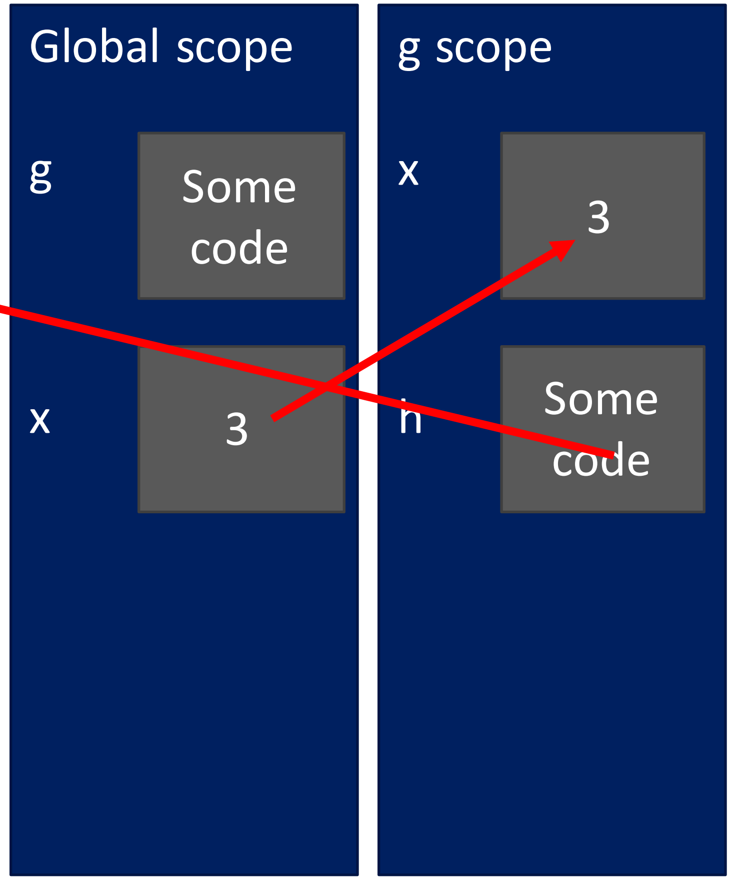
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```

Some code



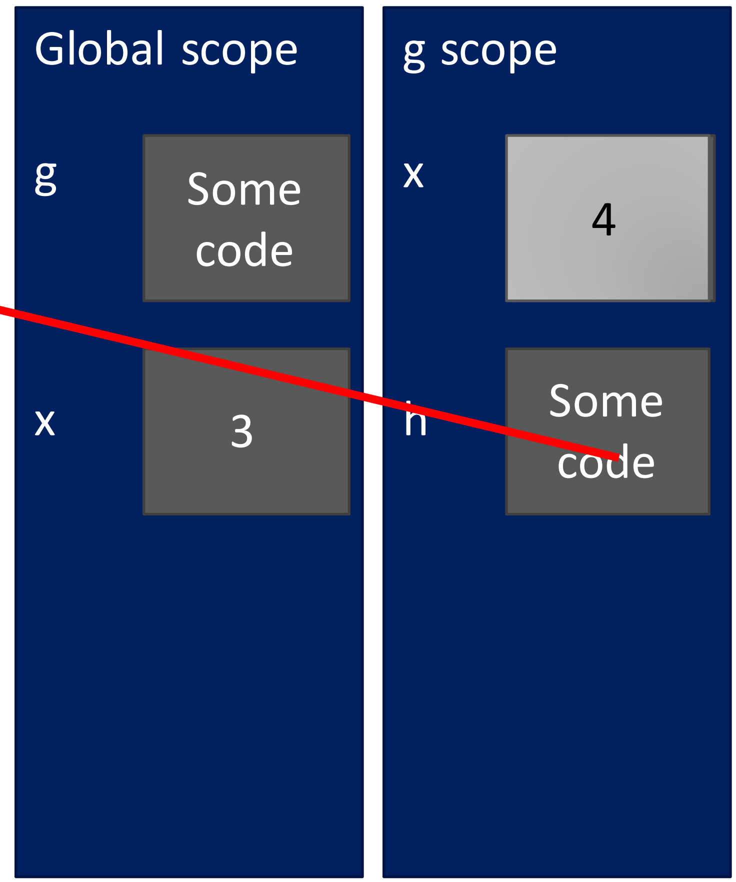
SCOPE DETAILS

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



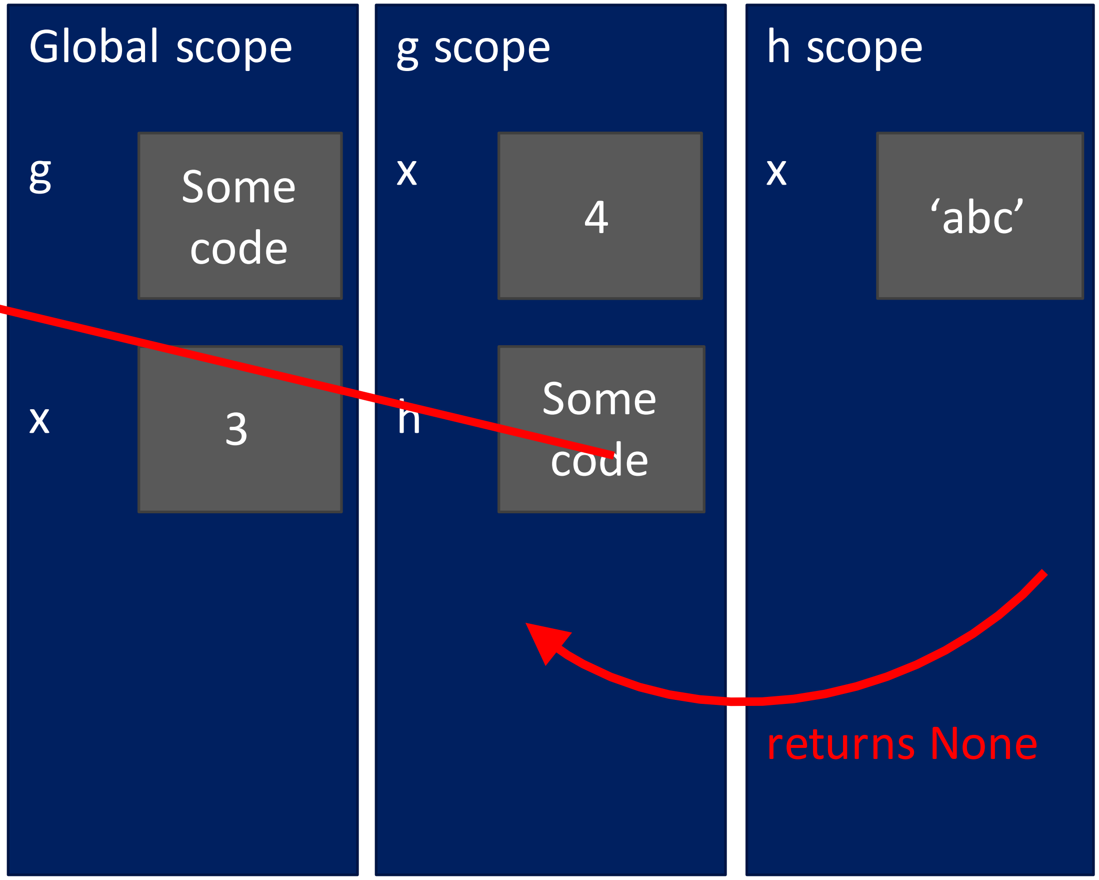
SCOPE DETAILS

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



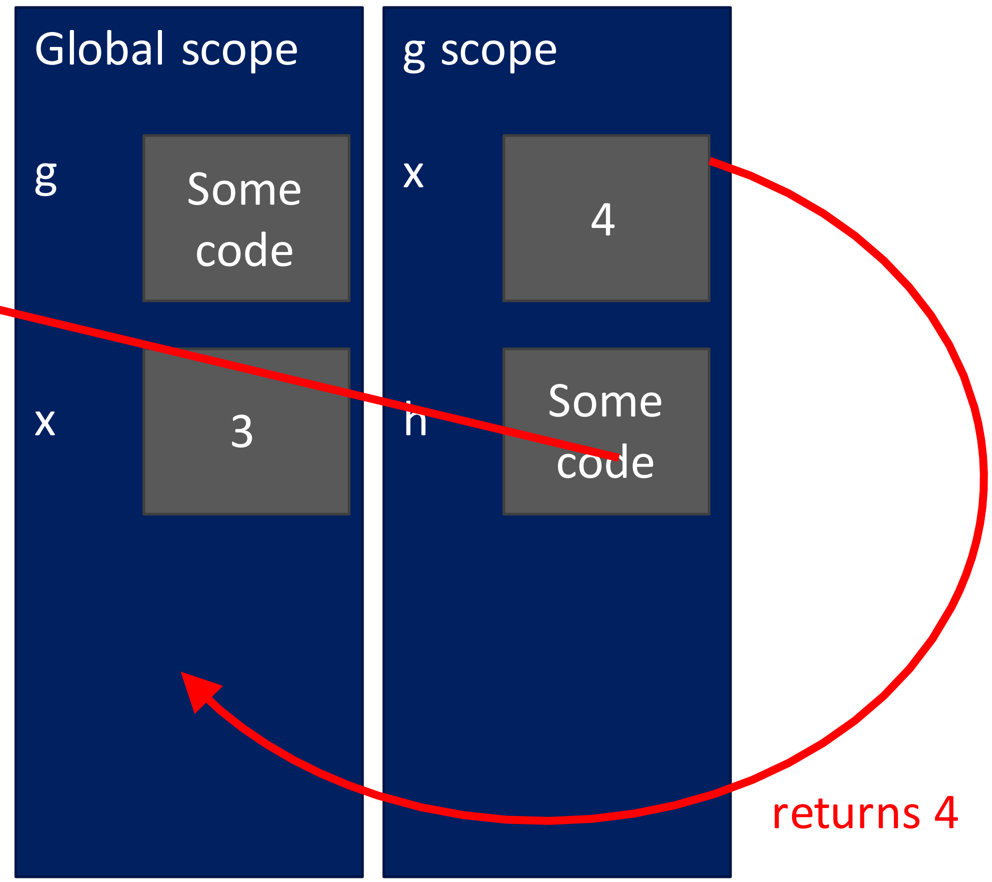
SCOPE DETAILS

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



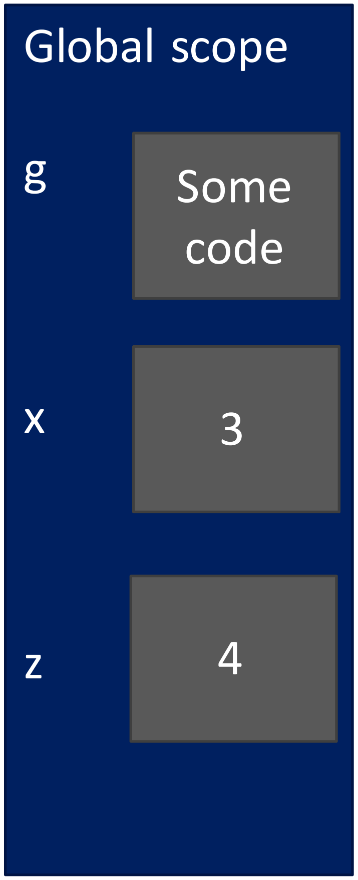
SCOPE DETAILS

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



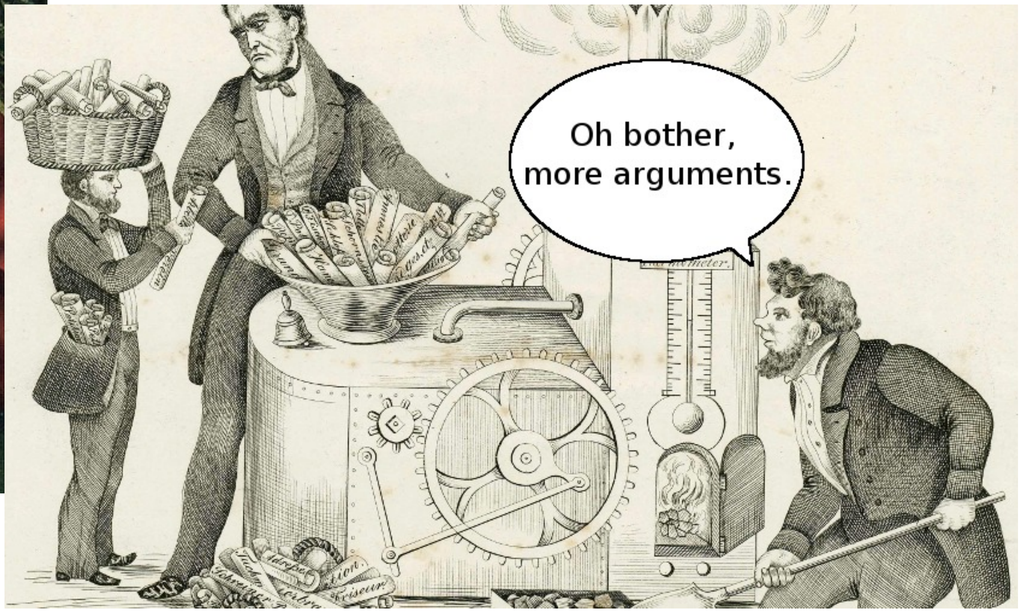
SCOPE DETAILS

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



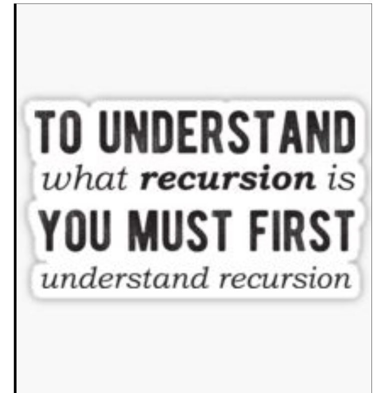
DECOMPOSITION & ABSTRACTION

- powerful together
- code can be used many times but only has to be debugged once!

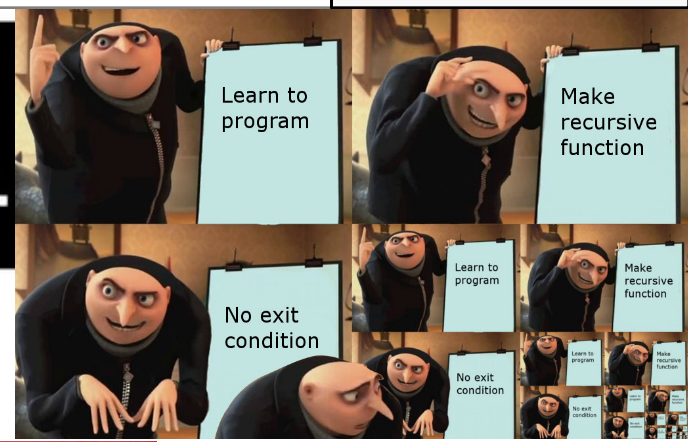


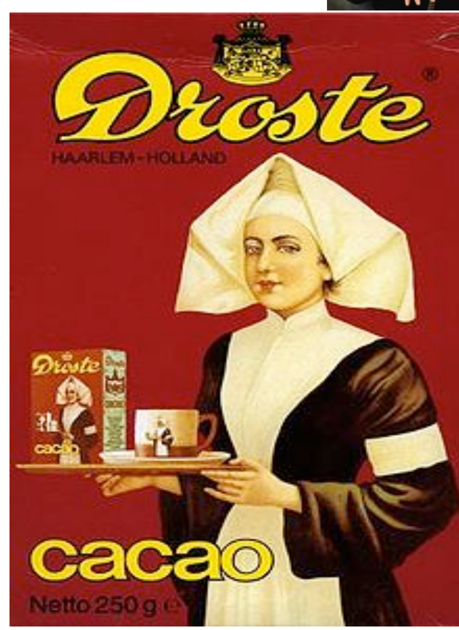
Five Minute Break

RECURSION



Recursion is the process of repeating items in a self-similar way.



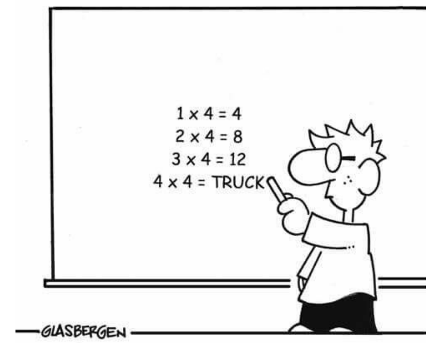


"mise en abyme"
Or
"Droste effect"
(1904)

ITERATIVE ALGORITHMS SO FAR

- looping constructs (`while` and `for` loops) lead to **iterative** algorithms
- can capture computation in a set of **state variables** that update, based on a set of rules, on each iteration through loop

MULTIPLICATION – ITERATIVE SOLUTION



- "multiply `a` * `b`" is equivalent to "add `a` to itself `b` times"
- capture **state** by
  - an **iteration** number (`i`) starts at b
    `i` ← `i-1` and stop when 0
  - a current **value of computation** (`result`) starts at 0
    `result` ← `result + a`

Update rules

`a + a + a + a + … + a`

i result: 0
i result: a
i result: 2a
i result: 3a
i result: 4a

```
def mult_iter(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

iteration

current value of computation, running sum

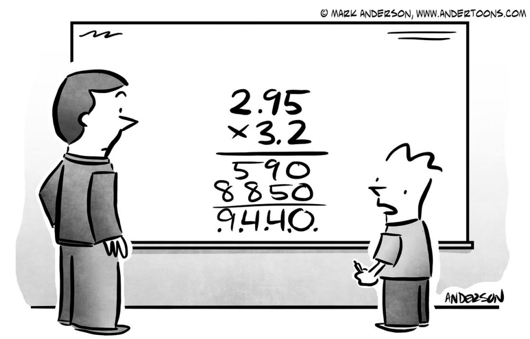
current value of iteration variable

Code we would write to capture iteration

Wrap inside a function, with return

Parameters set values for computation

MULTIPLICATION – RECURSIVE SOLUTION



“I forgot where to put the decimal, so I figured I’d cover all the bases.”

- **recursive step**
  - think how to reduce problem to a **simpler/smaller version** of same problem
- **base case**
  - keep reducing problem until reach a simple case that can be **solved directly**
  - when b = 1, a*b = a

$$\boxed{a*b} = \underbrace{a + a + a + a + \ldots + a}_{\text{b times}}$$

$$= a + \underbrace{a + a + a + \ldots + a}_{\text{b-1 times}}$$

$$= a + \boxed{a * (b-1)}$$

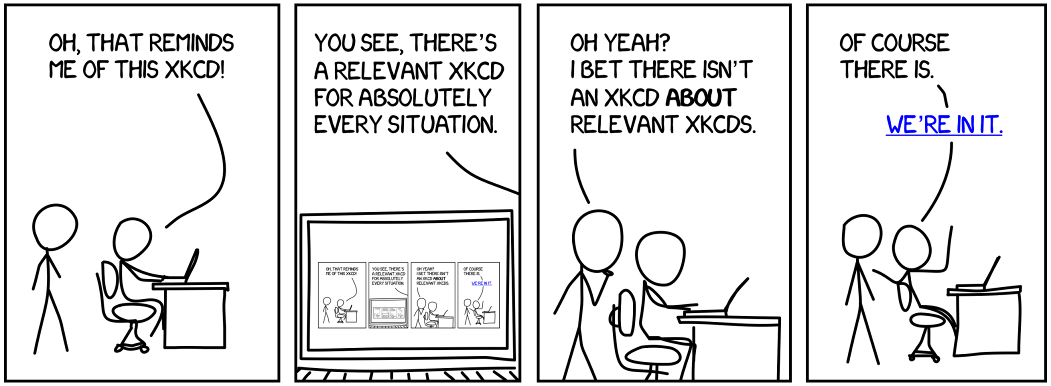
recursive reduction

```
def mult(a, b):
    if b == 1:
        return a
    else:
        return a + mult(a, b-1)
```

base case

recursive step

WHAT IS RECURSION?



- Algorithmically: a way to design solutions to problems by **divide-and-conquer** or **decrease-and-conquer**
  - reduce a problem to simpler versions of the same problem or problems that can be solved directly
- Semantically: a programming technique where a **function calls itself**
  - in programming, goal is to NOT have infinite recursion
    - must have **1 or more base cases** that are easy to solve directly
    - must solve the same problem on **some other input** with the goal of simplifying the larger input problem, ending at base case

FACTORIAL


```
n! = n*(n-1)*(n-2)*(n-3)* … * 1
```

- for what `n` do we know the factorial?

  n = 1 → `if n == 1:`

  `return 1` base case

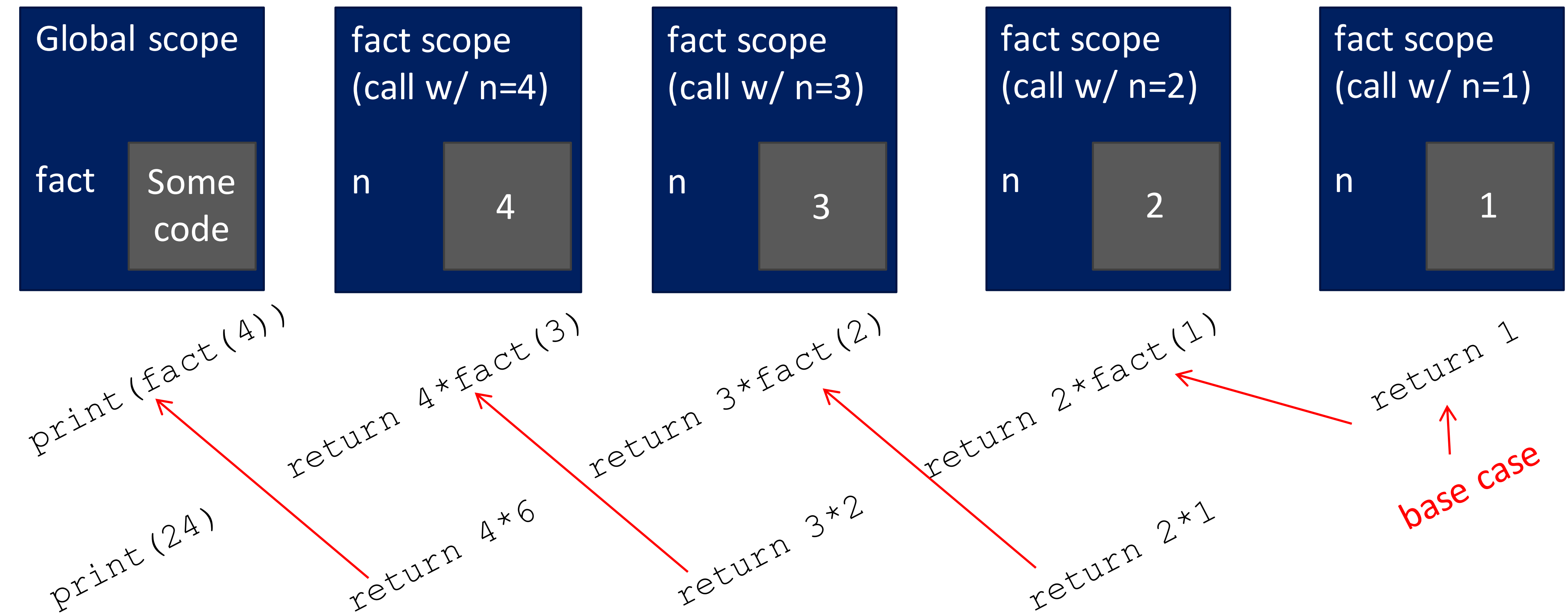
- how to reduce problem? Rewrite in terms of something simpler to reach base case

  n*(n-1)! → `else:`

  `return n*factorial(n-1)` recursive step

RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



YOUR TURN


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

If we evaluate fact(4), how many times is the procedure fact called?

A) 0

B) 1

C) 2

D) 3

E) 4

F) 5

G) infinitely many

SOME OBSERVATIONS

- each recursive call to a function creates its **own scope/environment**
- **bindings of variables** in a scope are not changed by recursive call
- flow of control passes back to **previous scope** once function call returns value

using the same variable names but they are different objects in separate scopes

ITERATION vs. RECURSION

```python
def factorial_iter(n):
    prod = 1
    for i in range(1,n+1):
        prod *= i
    return prod
```

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

This version is much more Pythonic!

- recursion may be simpler, more intuitive
- recursion may be efficient from programmer POV
- recursion may not be efficient from computer POV

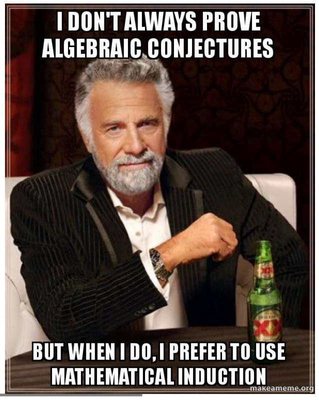
There is a way to implement recursive call in the Python evaluator (called tail recursion) that is very efficient

INDUCTIVE REASONING

- how do we know that our code will work (i.e. stop with right answer)?
- for iterative code (loops) we can reason using a decrementing function
- just use size of b in this case
- `mult_iter` terminates because b is initially positive, and decreases by 1 each time around loop; thus must eventually become less than 1
- correct value is computed since add b instances of a

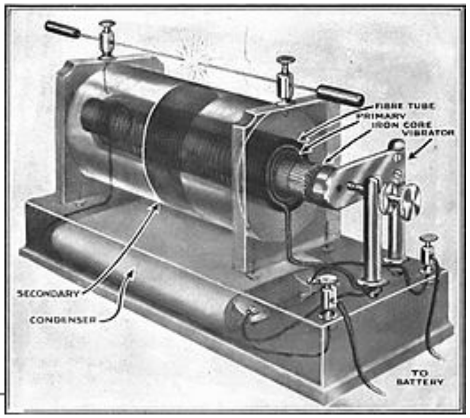
```
def mult_iter(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

MATHEMATICAL INDUCTION



- to prove a statement indexed on integers is true for all values of n:
  - prove it is true when n is smallest value (e.g. n = 0 or n = 1)
  - then prove that if it is true for all values up to n, one can show that it must be true for n+1

EXAMPLE OF INDUCTION



- $0 + 1 + 2 + 3 + \ldots + n = (n(n+1))/2$
- Proof:
  - if n = 0, then LHS is 0 and RHS is 0*1/2 = 0, so true
  - assume true for all values up to n, then need to show that

    $$\underbrace{0 + 1 + 2 + \ldots + n} + (n+1) = ((n+1)(n+2))/2$$

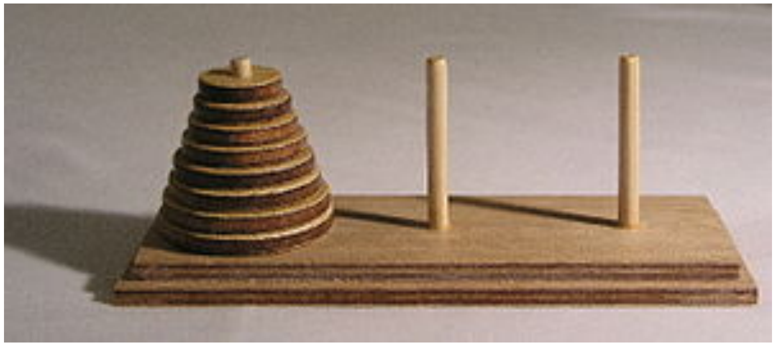
    - LHS is $n(n+1)/2 + (n+1)$ by assumption that property holds for problem of size n or smaller
    - this becomes, by algebra, $((n+1)(n+2))/2$
  - hence expression holds for all n >= 0

INDUCTIVE REASONING

- how do we know that our recursive code will work (i.e. stop with right answer)?
  - use **induction**
- `mult` called with b = 1 has no recursive call and stops
- `mult` called with b > 1 makes a recursive call with a smaller version of b; so eventually will halt when b == 1
- by induction, if simpler version of recursive call returns correct value, then so does current call

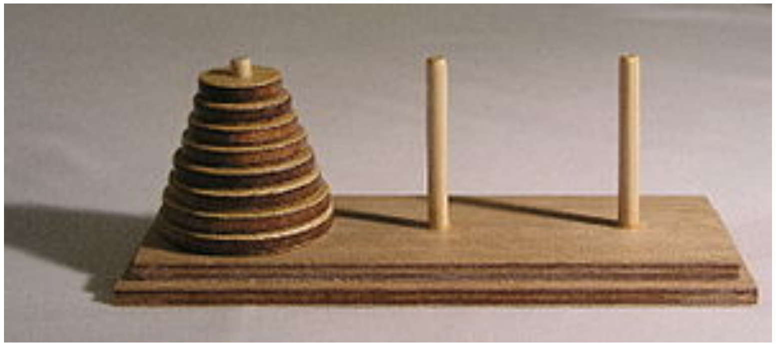
```
def mult(a, b):
    if b == 1:
        return a
    else:
        return a + mult(a, b-1)
```

TOWERS OF HANOI

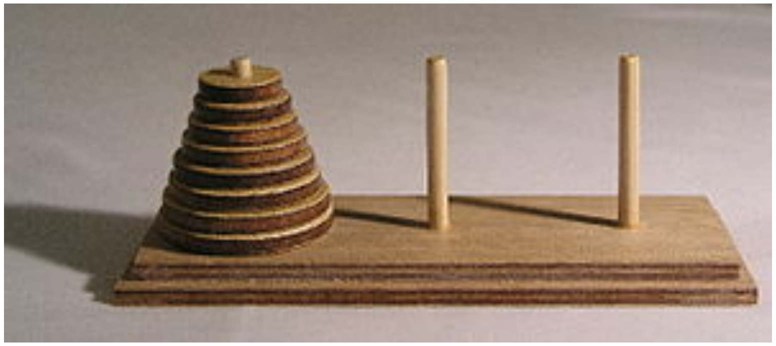


- The story:
  - 3 tall spikes
  - stack of 64 different sized discs – start on one spike, ordered from smallest to largest
  - need to move stack to second spike (at which point universe ends)
  - only move one disc at a time, larger disc can't cover smaller disc

TOWERS OF HANOI



- having seen a set of examples of different sized stacks, how would you write a program to print out the right set of moves?
- **Think recursively!**
  - solve a smaller problem
  - solve a basic problem
  - solve a smaller problem

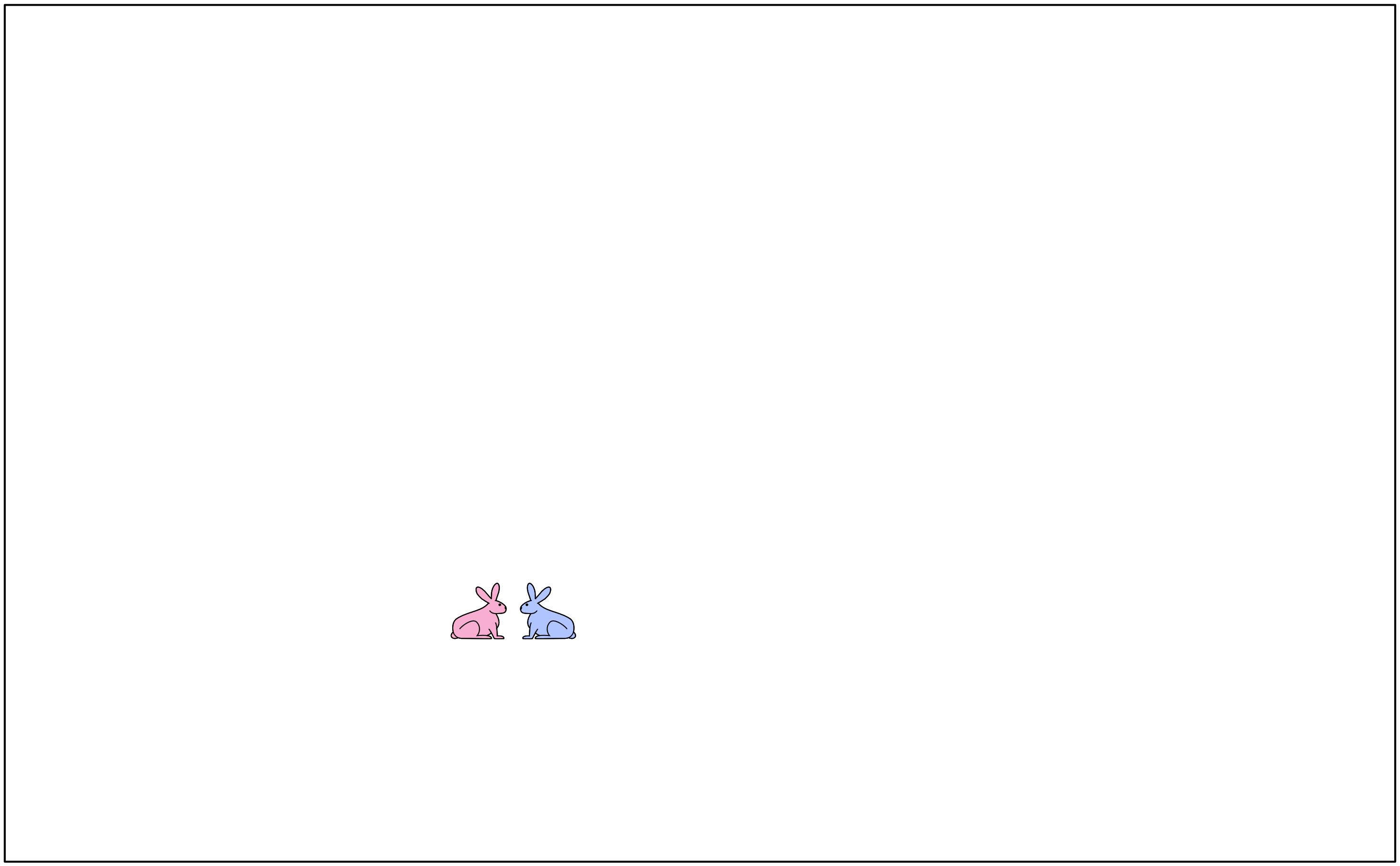


```python
def printMove(fr, to):
    print('move from ' + str(fr) + ' to ' + str(to))


def Towers(n, fr, to, spare):
    if n == 1:
        printMove(fr, to)
    else:
        Towers(n-1, fr, spare, to)
        Towers(1, fr, to, spare)
        Towers(n-1, spare, to, fr)
```

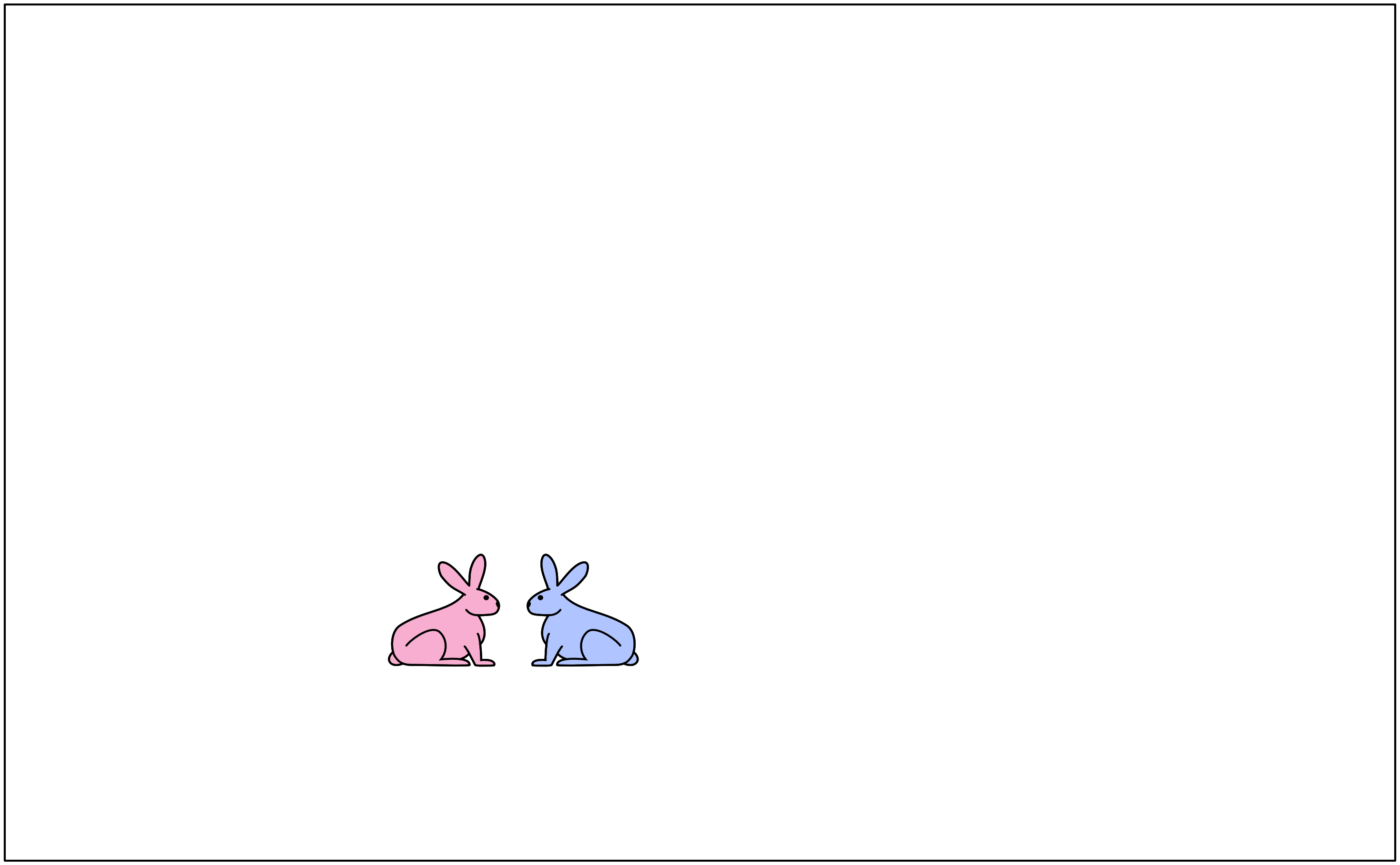
BTW, if move a disc every millisecond, will take 5.8 X 10^8 years to complete

RECURSION WITH MULTIPLE BASE CASES

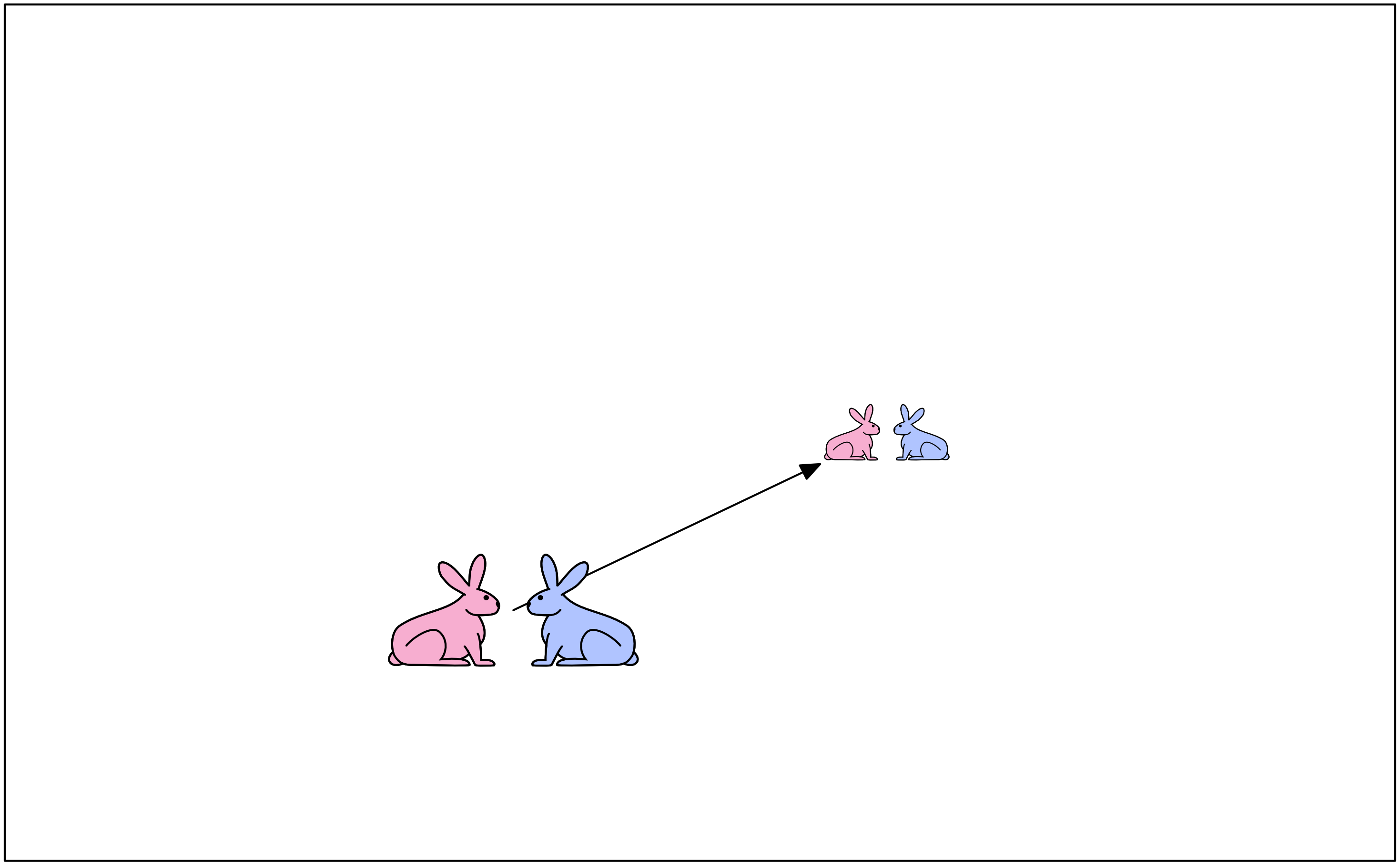
- Fibonacci numbers
  - Leonardo of Pisa (aka Fibonacci) modeled the following challenge
    - newborn pair of rabbits (one female, one male) are put in a pen
    - rabbits mate at age of one month
    - rabbits have a one month gestation period
    - assume rabbits never die, that female always produces one new pair (one male, one female) each month from its second month on.
    - how many female rabbits are there at the end of one year?



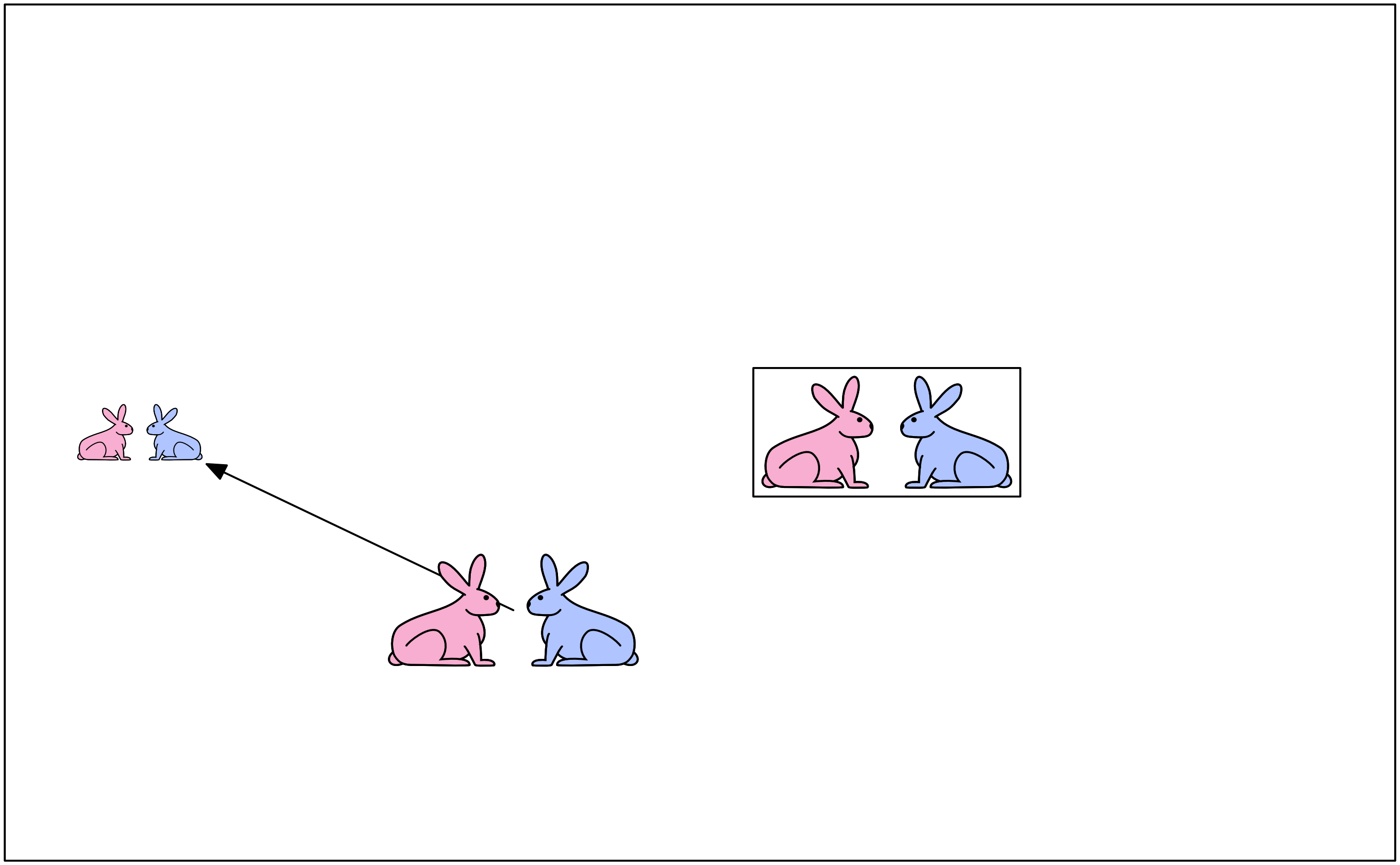
Demo courtesy of Prof. Denny Freeman and Adam Hartz



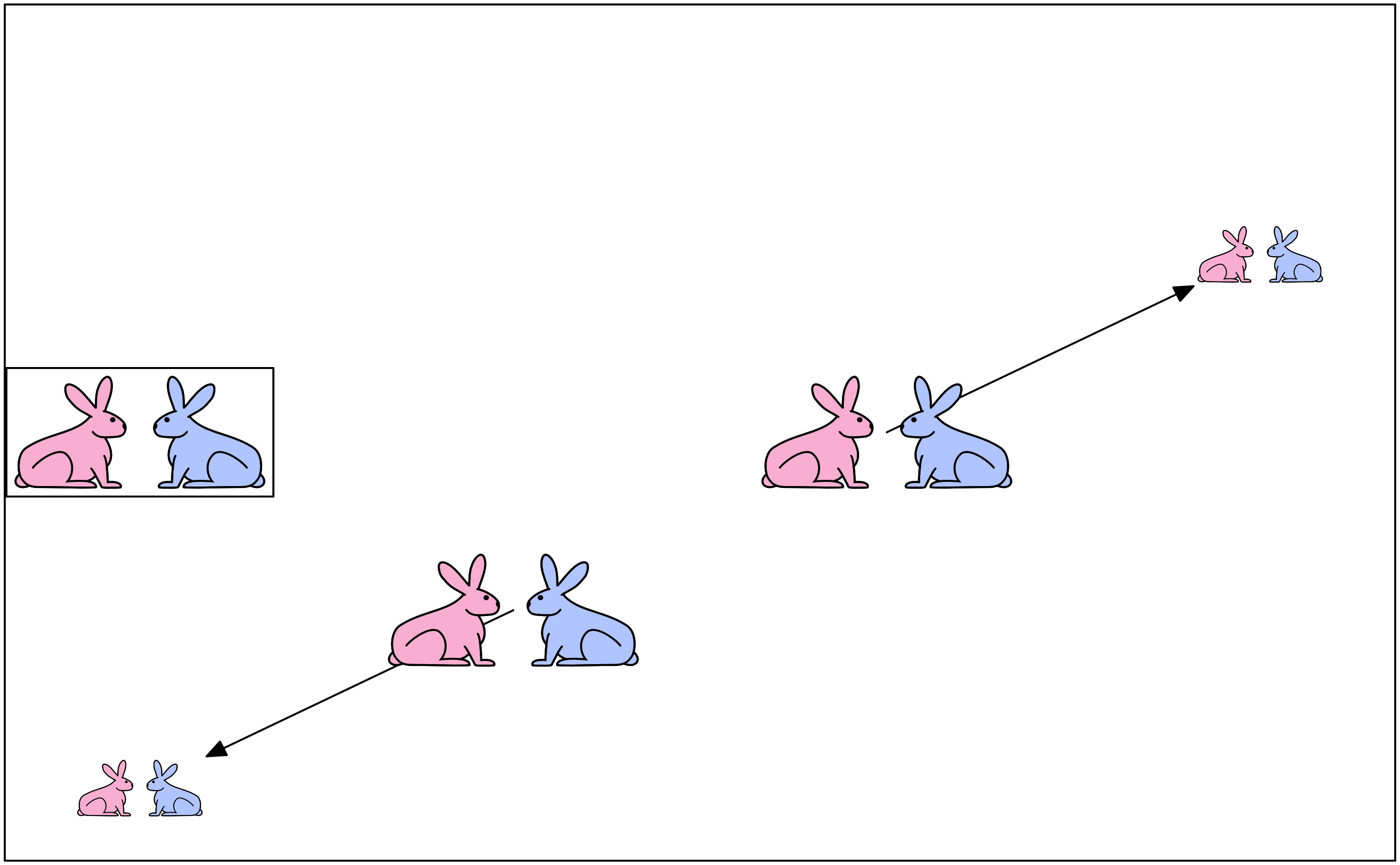
Demo courtesy of Prof. Denny Freeman and Adam Hartz



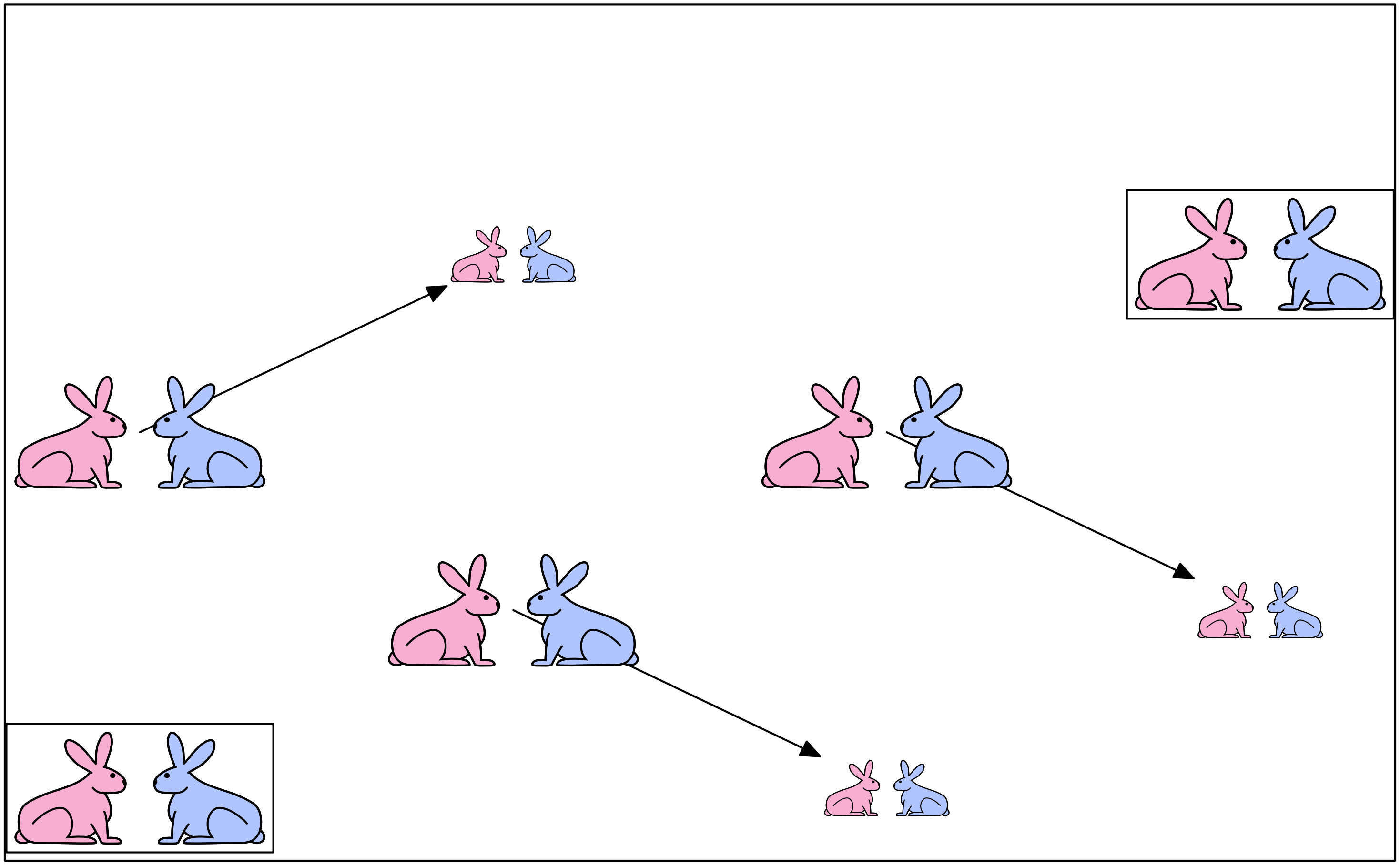
Demo courtesy of Prof. Denny Freeman and Adam Hartz



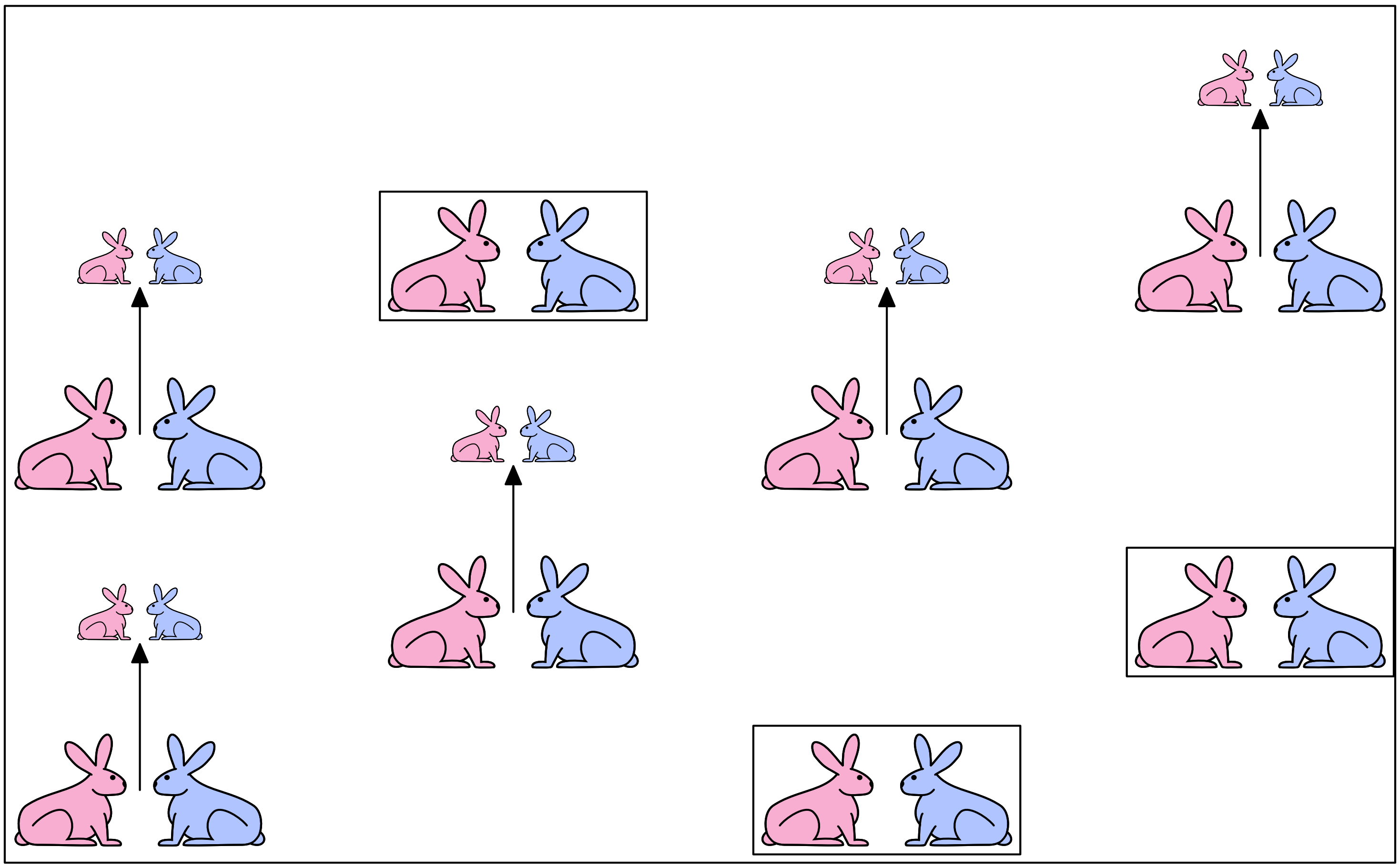
Demo courtesy of Prof. Denny Freeman and Adam Hartz



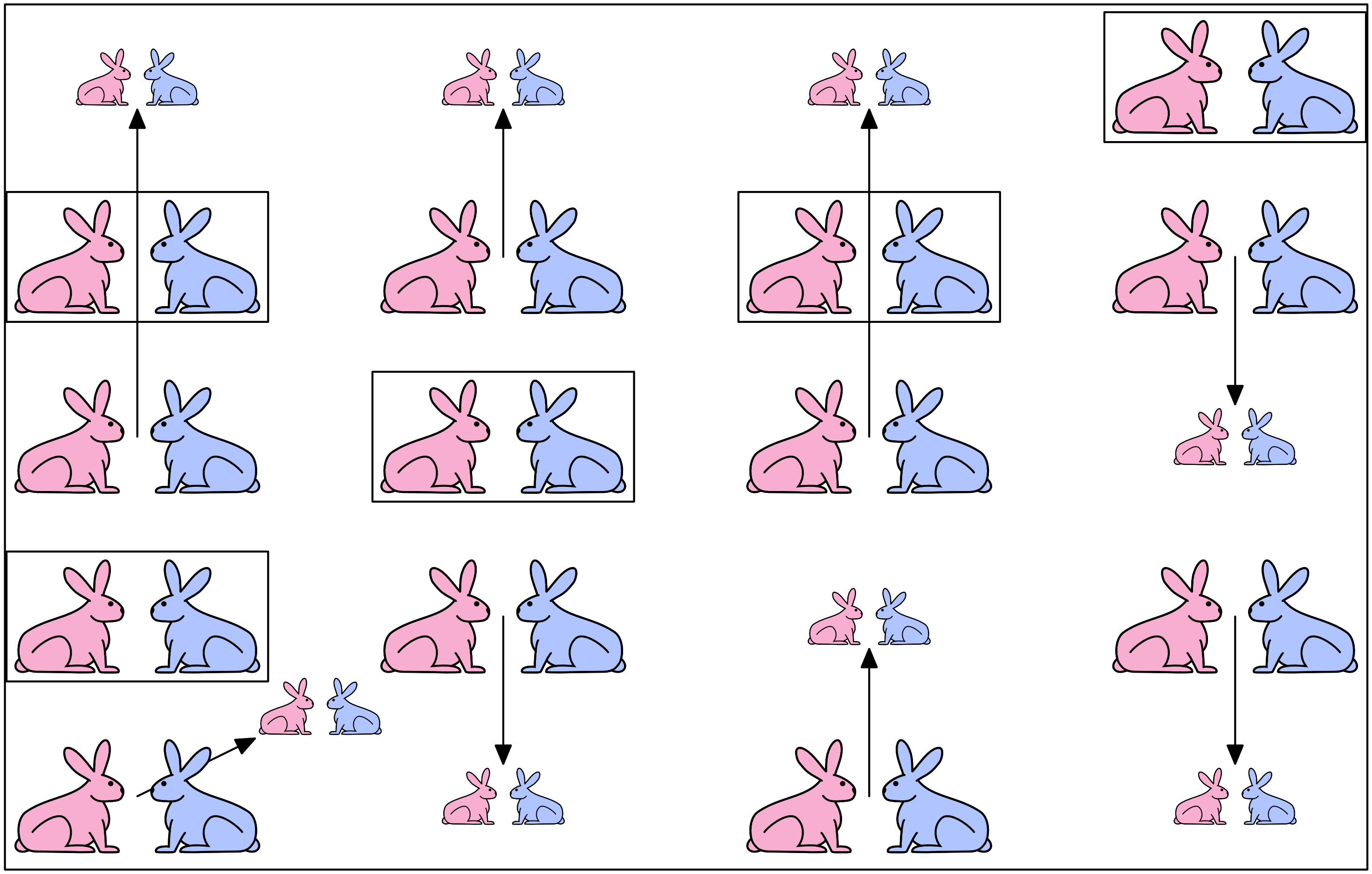
Demo courtesy of Prof. Denny Freeman and Adam Hartz



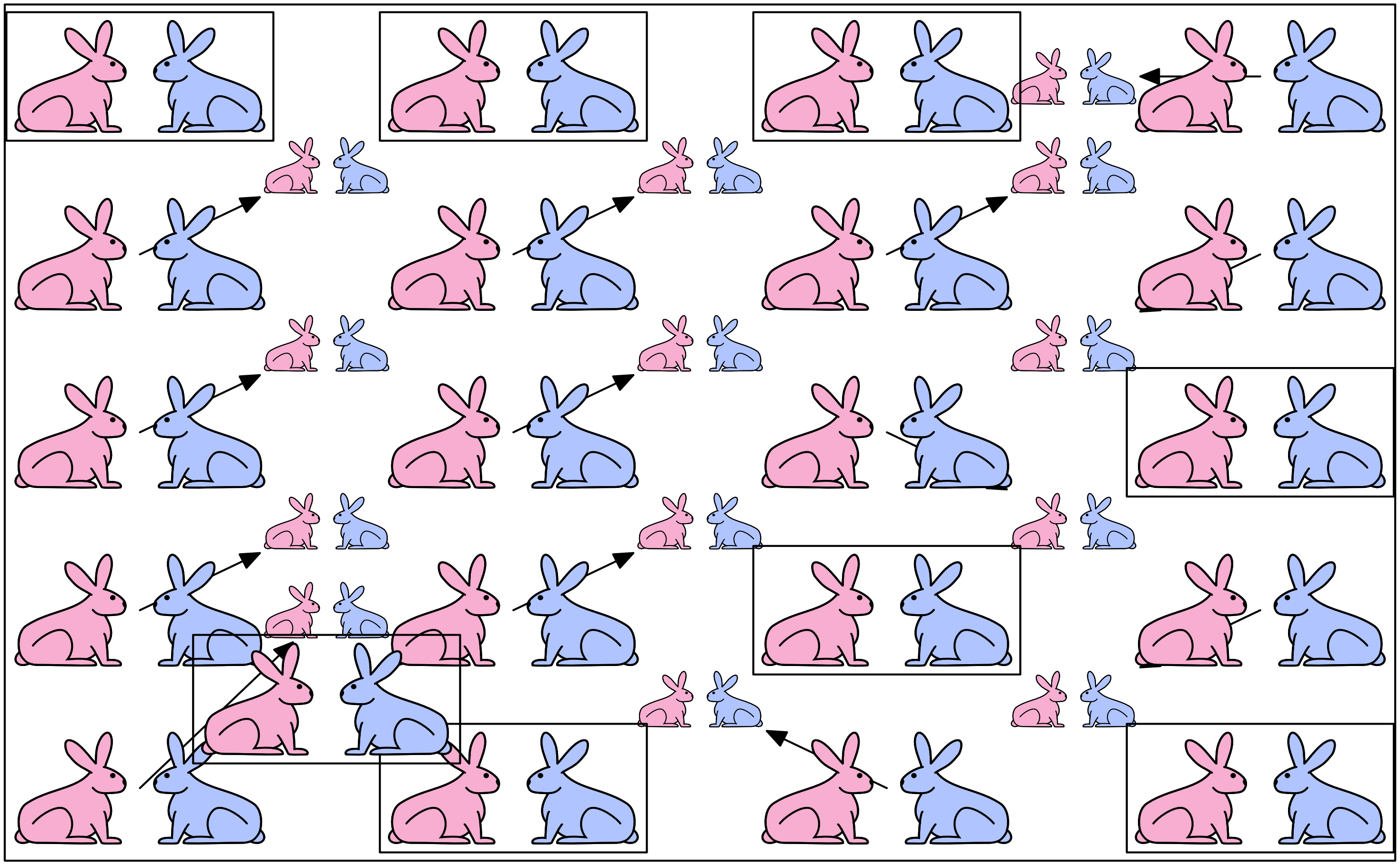
Demo courtesy of Prof. Denny Freeman and Adam Hartz



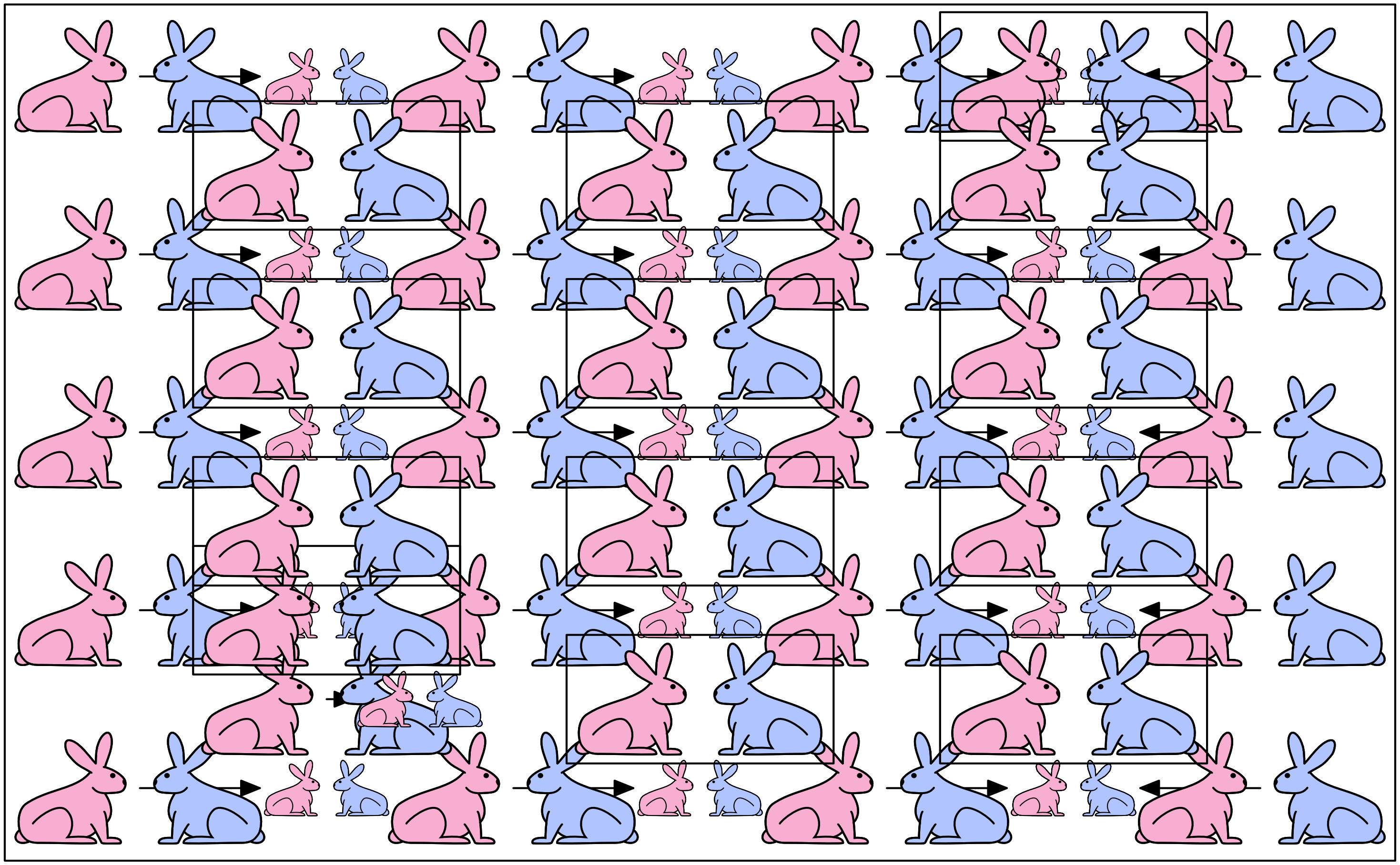
Demo courtesy of Prof. Denny Freeman and Adam Hartz



Demo courtesy of Prof. Denny Freeman and Adam Hartz

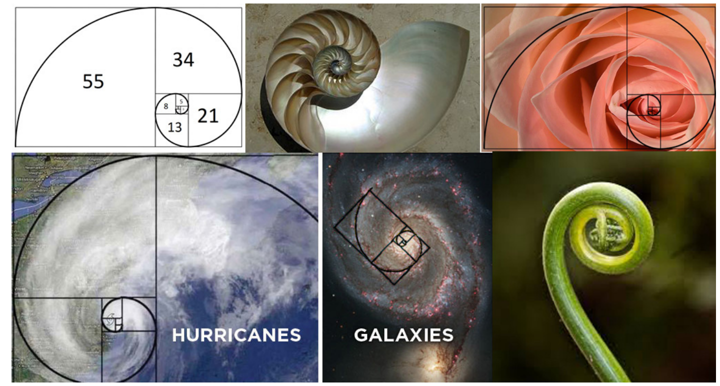


Demo courtesy of Prof. Denny Freeman and Adam Hartz



Demo courtesy of Prof. Denny Freeman and Adam Hartz

FIBONACCI



After one month (call it 0) – 1 female

After second month – still 1 female (now pregnant)

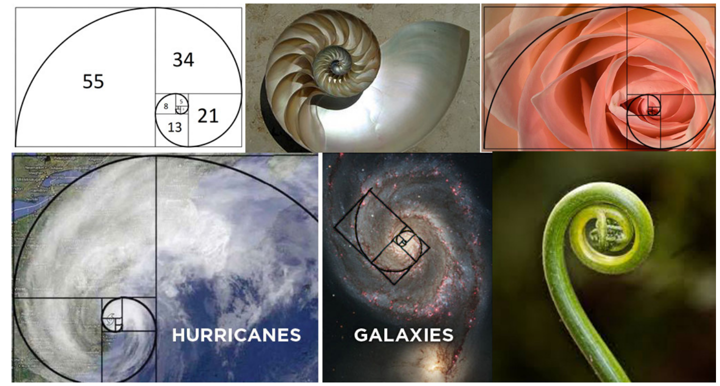
After third month – two females, one pregnant, one not

In general, females(n) = females(n-1) + females(n-2)

- Every female alive at month n-2 will produce one female in month n;
- These can be added those alive in month n-1 to get total alive in month n

Month	Females
0	1

FIBONACCI



- Base cases:
  - Females(0) = 1
  - Females(1) = 1
- Recursive case
  - Females(n) = Females(n-1) + Females(n-2)

This many does alive at time n-1

This many does alive at time n-2; each pregnant next month, so this many new does whelped at time n

FIBONACCI RECURSIVE CODE (MULTIPLE BASE CASES)

```
def fib(x):
    """assumes x an int >= 0
       returns Fibonacci of x"""
    if x == 0 or x == 1:
        return 1
    else:
        return fib(x-1) + fib(x-2)
```

TAKE HOME MESSAGES

- procedures (or functions) allow us to suppress detail and capture computation within a black box
- iteration works well with methods that are characterized by state variables
- recursion is a powerful tool that works well when solving one problem reduces to solving a simpler version of the same problem, plus some simple operations