DECOMPOSITION, ABSTRACTION, FUNCTIONS

HOW DO WE WRITE CODE?

- so far…
  - covered language mechanisms
  - know how to write different files for each computation
  - each file is some piece of code
  - each code is a sequence of instructions
- problems with this approach
  - easy for small-scale problems
  - messy for larger problems
  - hard to keep track of details
  - how do you know the right info is supplied to the right part of code

GOOD PROGRAMMING

- more code not necessarily a good thing
- measure good programmers by the amount of functionality
- introduce **functions**
- mechanism to achieve **decomposition** and **abstraction**

EXAMPLE -- PROJECTOR

http://www.myprojectorlamps.com/blog/wp-content/uploads/Dell-1610HD-Projector.jpg

- a projector is a black box
- don't know how it works
- know the interface: input/output
- connect any electronics to it that can communicate with that input
- black box somehow converts image from input source to a wall, magnifying it
- **ABSTRACTION IDEA**: do not need to know how projector works to use it

EXAMPLE -- PROJECTOR

- projecting large image for Olympics decomposed into separate tasks for separate projectors
- each projector takes input and produces separate output
- all projectors work together to produce larger image
- **DECOMPOSITION IDEA**: different devices work together to achieve an end goal

APPLY THESE IDEAS TO PROGRAMMING

- **DECOMPOSITION**
  - Break problem into different, self-contained, pieces
- **ABSTRACTION**
  - Suppress details of method to compute something from use of that computation

CREATE STRUCTURE with DECOMPOSITION

- in example, separate devices
- in programming, divide code into **modules**
  - are **self-contained**
  - used to **break up** code
  - intended to be **reusable**
  - keep code **organized**
  - **keep code coherent**
- this lecture, achieve decomposition with **functions**
- in a few weeks, achieve decomposition with **classes**

SUPPRESS DETAILS with ABSTRACTION

- in example, no need to know how to build a projector
- in programming, think of a piece of code as a **black box**
  - cannot see details
  - do not need to see details
  - do not want to see details
  - hide tedious coding details
- achieve abstraction with **function specifications** or **docstrings**

DECOMPOSITION & ABSTRACTION

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

FUNCTIONS

- write reusable piece/chunks of code, called **functions**
- functions are not run in a program until they are "**called**" or "**invoked**" in a program
- function characteristics:
  - has a **name**
  - has **parameters** (0 or more)
  - has a **docstring** (optional but recommended)
  - has a **body**

HOW TO WRITE and CALL/INVOKE A FUNCTION

keyword

name

parameters or arguments

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

Specification, docstring

body

```
is_even(3)
```

later in the code, you call the function using its name and 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("hi")
    return i%2 == 0
```

evaluate some expressions

keyword

expression to evaluate and return

VARIABLE SCOPE

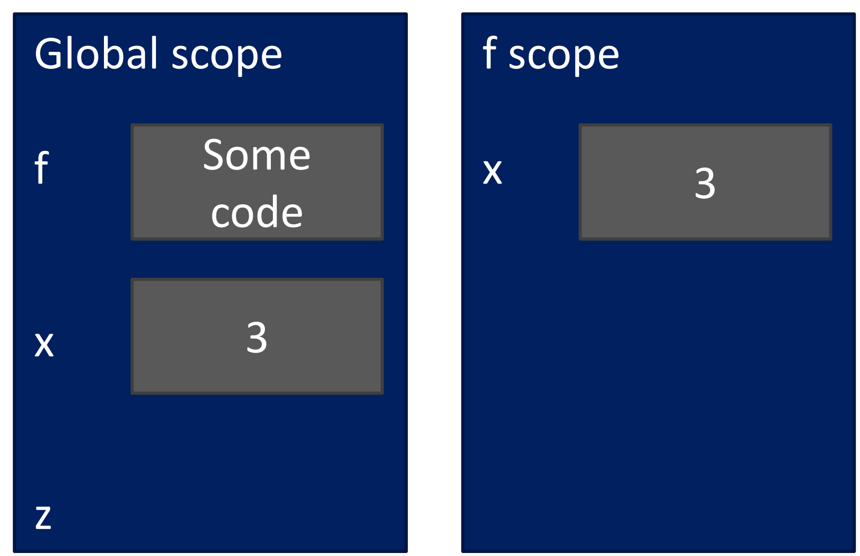
- **formal parameter** gets bound to the value of **actual parameter** when function is called
- new **scope/frame/environment** created when enter a function
- **scope** is mapping of names to objects

formal parameter

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

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

actual parameter

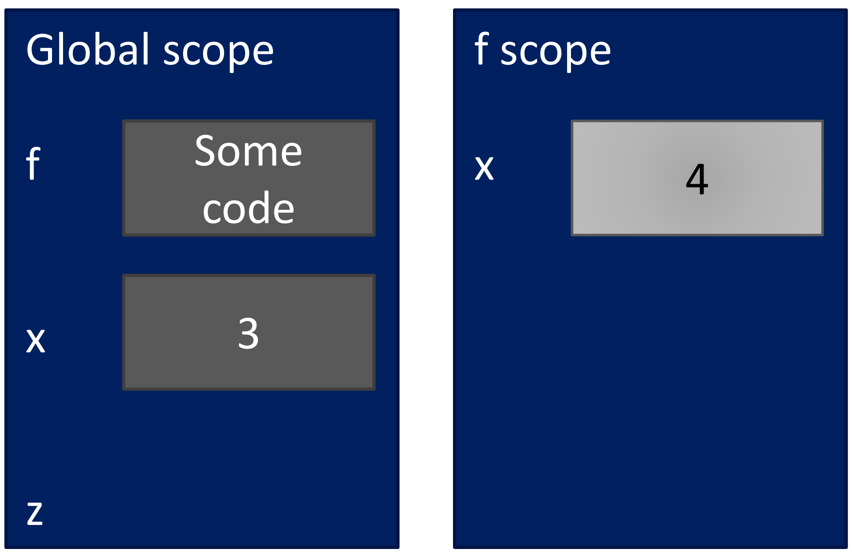
call to f, before body evaluated



VARIABLE SCOPE

- **formal parameter** gets bound to the value of **actual parameter** when function is called
- new **scope/frame/environment** created when enter a function
- **scope** is mapping of names to objects

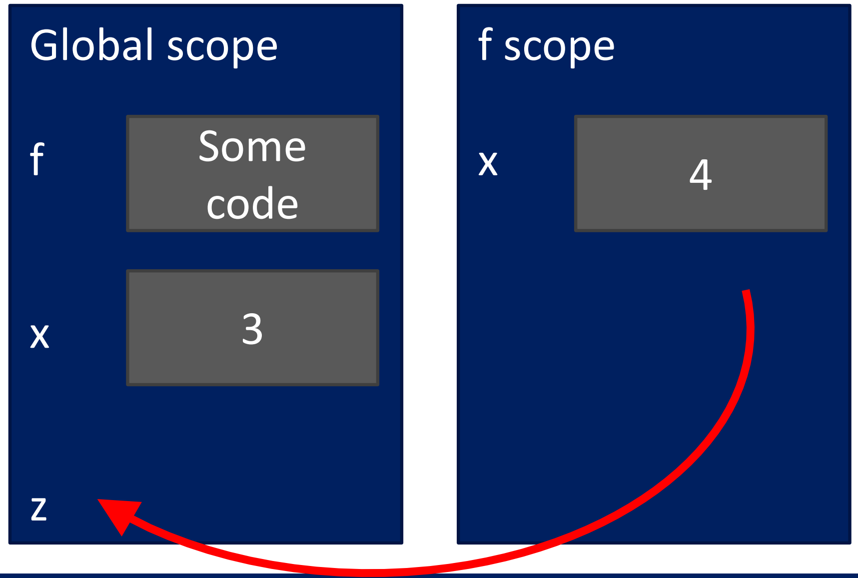
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x
x = 3
z = f( x )
```



VARIABLE SCOPE

- **formal parameter** gets bound to the value of **actual parameter** when function is called
- new **scope/frame/environment** created when enter a function
- **scope** is mapping of names to objects

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x
x = 3
z = f( x )
```

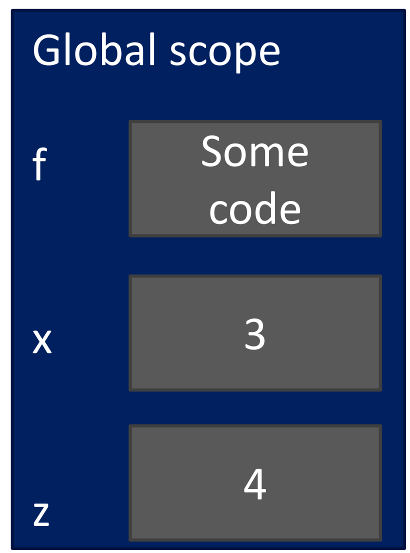


VARIABLE SCOPE

- **formal parameter** gets bound to the value of **actual parameter** when function is called
- new **scope/frame/environment** created when enter a function
- **scope** is mapping of names to objects

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x
x = 3
z = f( x )
```

binding of returned value to variable z



ONE WARNING IF NO `return` STATEMENT

```
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

`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

FUNCTIONS AS ARGUMENTS

- arguments 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(z):
    print('inside func_c’)
    return z()
print(func_a())
print(5 + func_b(2))
print(func_c(func_a))
```

call func_a, takes no parameters

call func_b, takes one parameter

call func_c, takes one parameter, another function

SCOPE EXAMPLE

- inside a function, **can access** a variable defined outside
- inside a function, **cannot modify** a variable defined outside

```
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

```
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

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

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

UnboundLocalError: local variable 'x' referenced before assignment

SCOPE EXAMPLE

- inside a function, **can access** a variable defined outside
- inside a function, **cannot modify** a variable defined outside

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

x = 5
f(2)
print x
```

```
def g(y):
    print x

x = 5
g(2)
print x
```

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

x = 5
h(2)
print x
```

x from global/main program scope

HARDER SCOPE EXAMPLE

IMPORTANT and TRICKY!

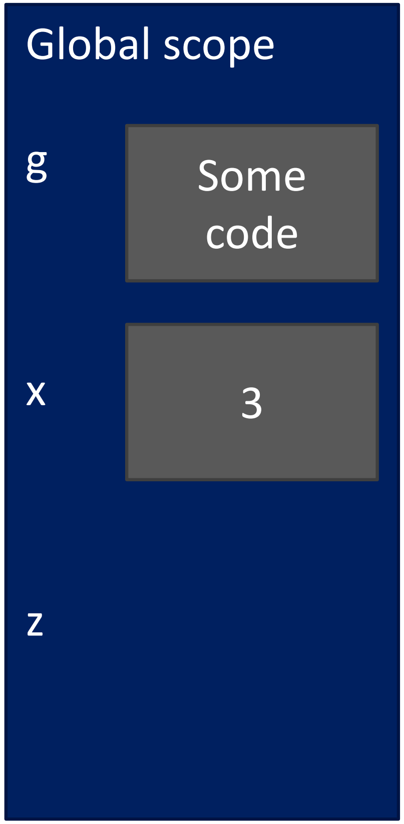
Python Tutor is your best friend to help sort this out!

http://www.pythontutor.com/

SCOPE DETAILS

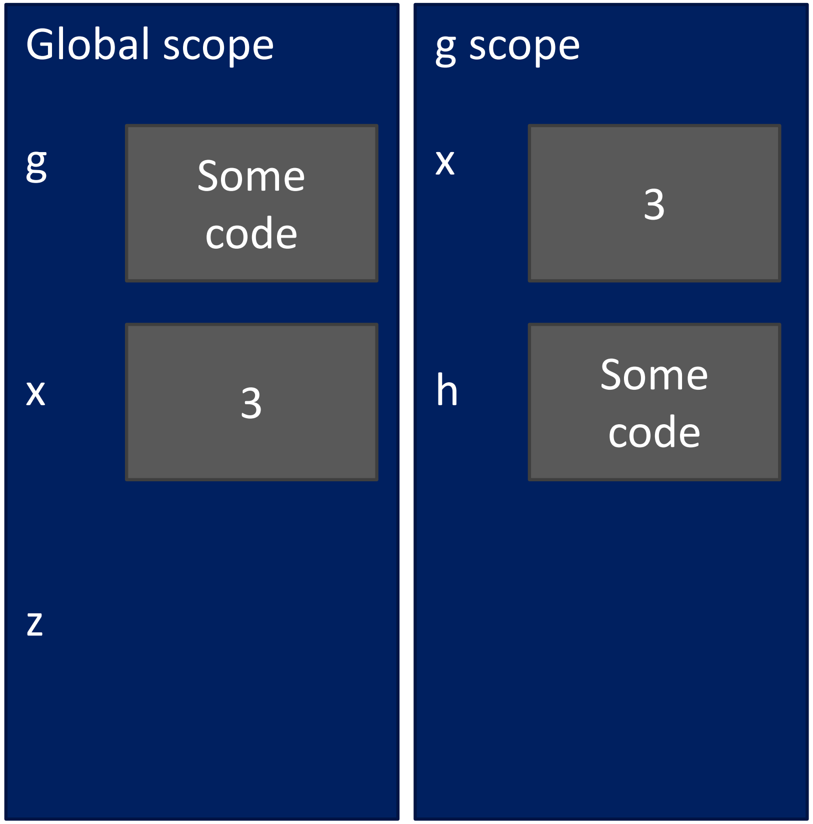
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('in g(x): x =', x)
    h()
    return x

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

Some code



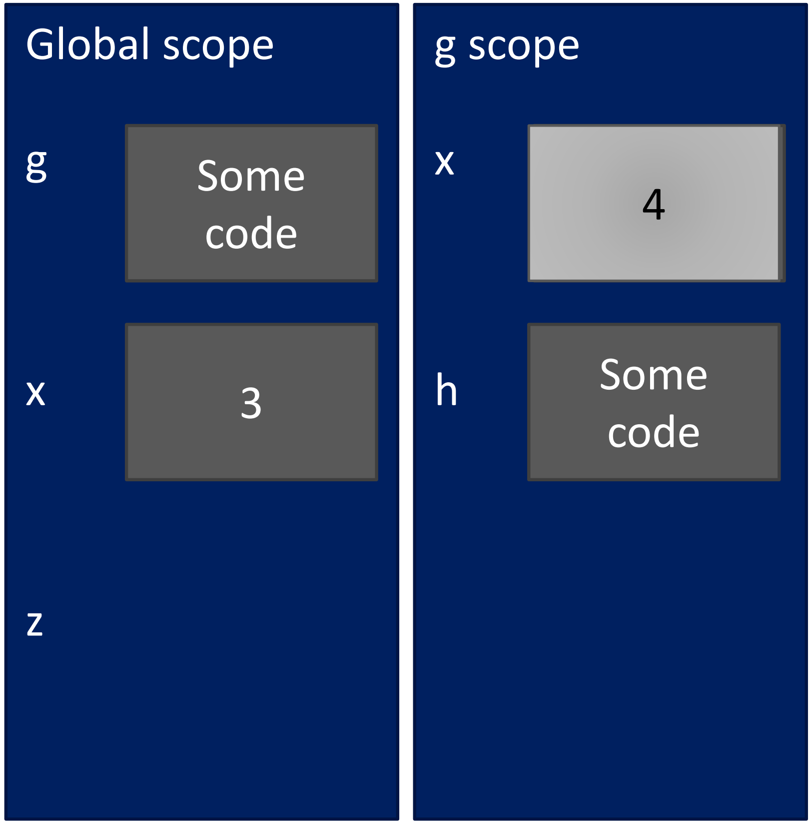
SCOPE DETAILS

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

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



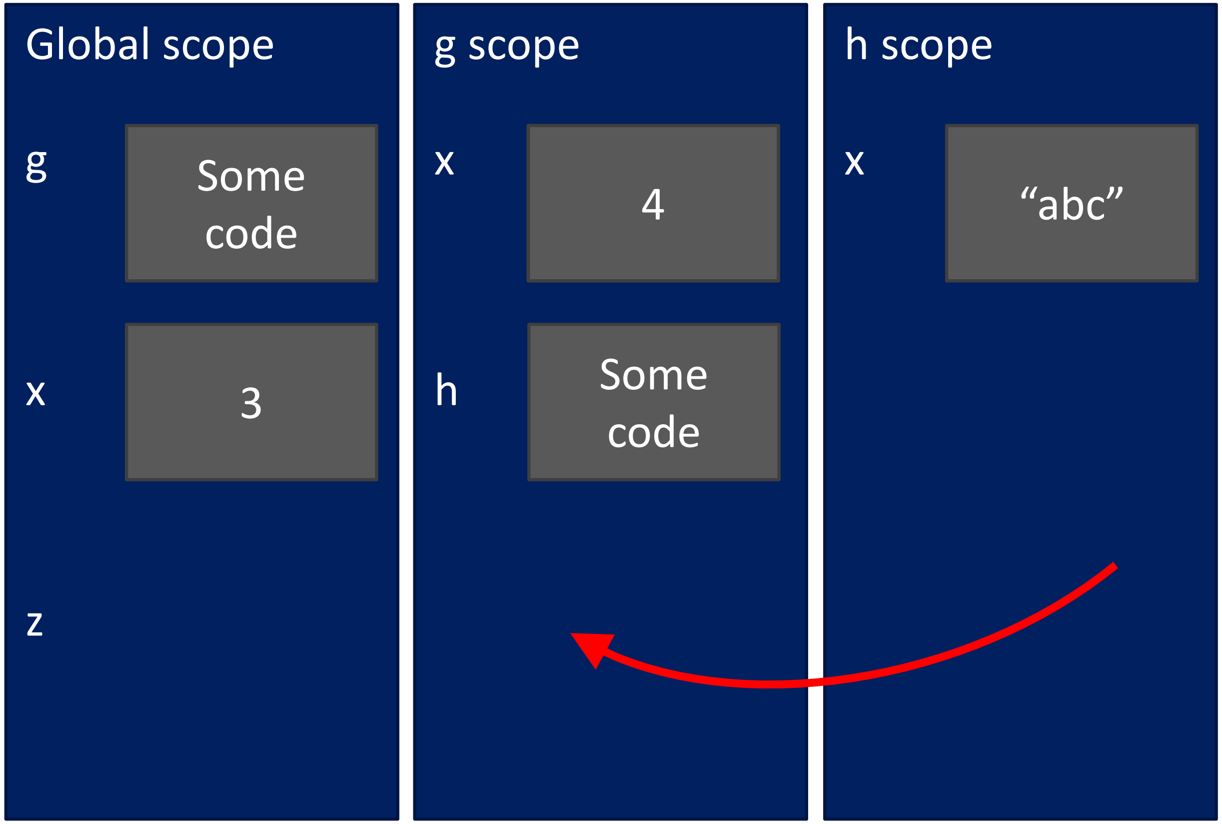
SCOPE DETAILS

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

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



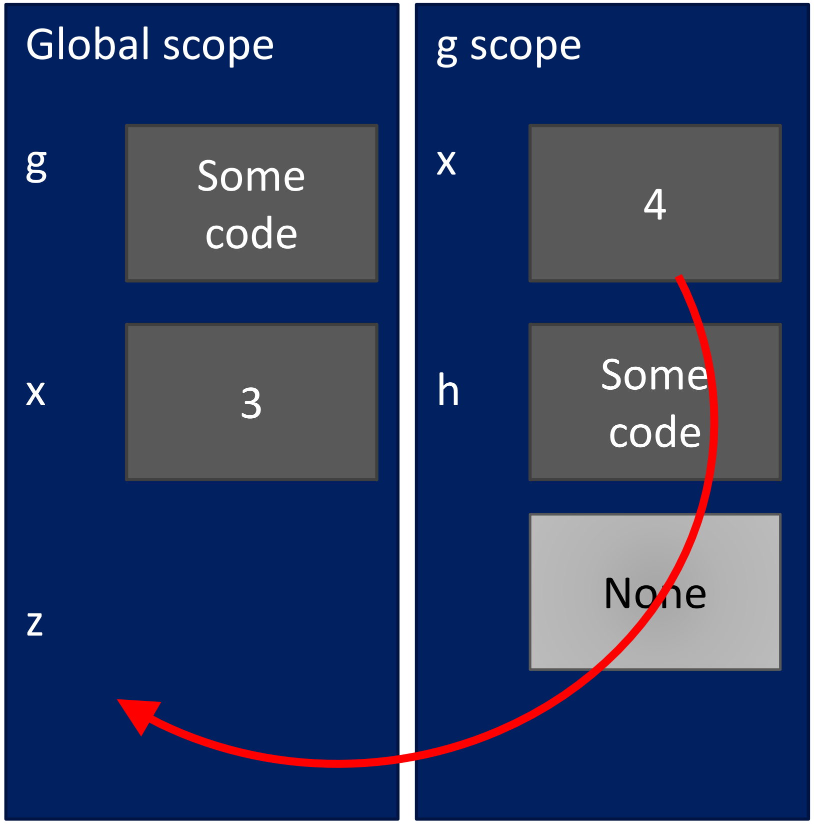
SCOPE DETAILS

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

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



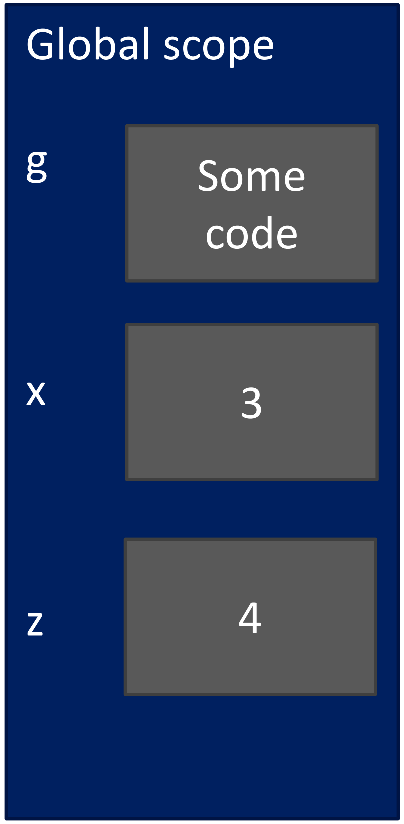
SCOPE DETAILS

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

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



SCOPE DETAILS

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

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



KEYWORD ARGUMENTS AND DEFAULT VALUES

- Simple function definition, if last argument is TRUE, then print lastName, firstName; else firstName, lastName

```
def printName(firstName, lastName, reverse):
    if reverse:
        print(lastName + ', ' + firstName)
    else:
        print(firstName, lastName)
```

KEYWORD ARGUMENTS AND DEFAULT VALUES

- Each of these invocations is equivalent

```
printName('Eric', 'Grimson', False)

printName('Eric', 'Grimson', reverse = False)

printName('Eric', lastName = 'Grimson', reverse = False)

printName(lastName = 'Grimson', firstName = 'Eric',
          reverse = False)
```

KEYWORD ARGUMENTS AND DEFAULT VALUES

- Can specify that some arguments have default values, so if no value supplied, just use that value

```
def printName(firstName, lastName, reverse = False):
    if reverse:
        print(lastName + ', ' + firstName)
    else:
        print(firstName, lastName)


printName('Eric', 'Grimson')
printName('Eric', 'Grimson', True)
```

SPECIFICATIONS

- a **contract** between the implementer of a function and the clients who will use it
  - **Assumptions:** conditions that must be met by clients of the function; typically constraints on values of parameters
  - **Guarantees:** conditions that must be met by function, providing it has been called in manner consistent with assumptions

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

is_even(3)
```

WHAT IS RECURSION

- a way to design solutions to problems by **divide-and-conquer or decrease-and-conquer**
- 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
  - must solve the same problem on **some other input** with the goal of simplifying the larger problem input

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 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)

    result ← result + a

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

iteration

current value of computation,
a running sum

current value of iteration variable

MULTIPLICATION – RECURSIVE SOLUTION

- **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

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

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

$$= a + a * (b-1)$$

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

base case

recursive step

FACTORIAL

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

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

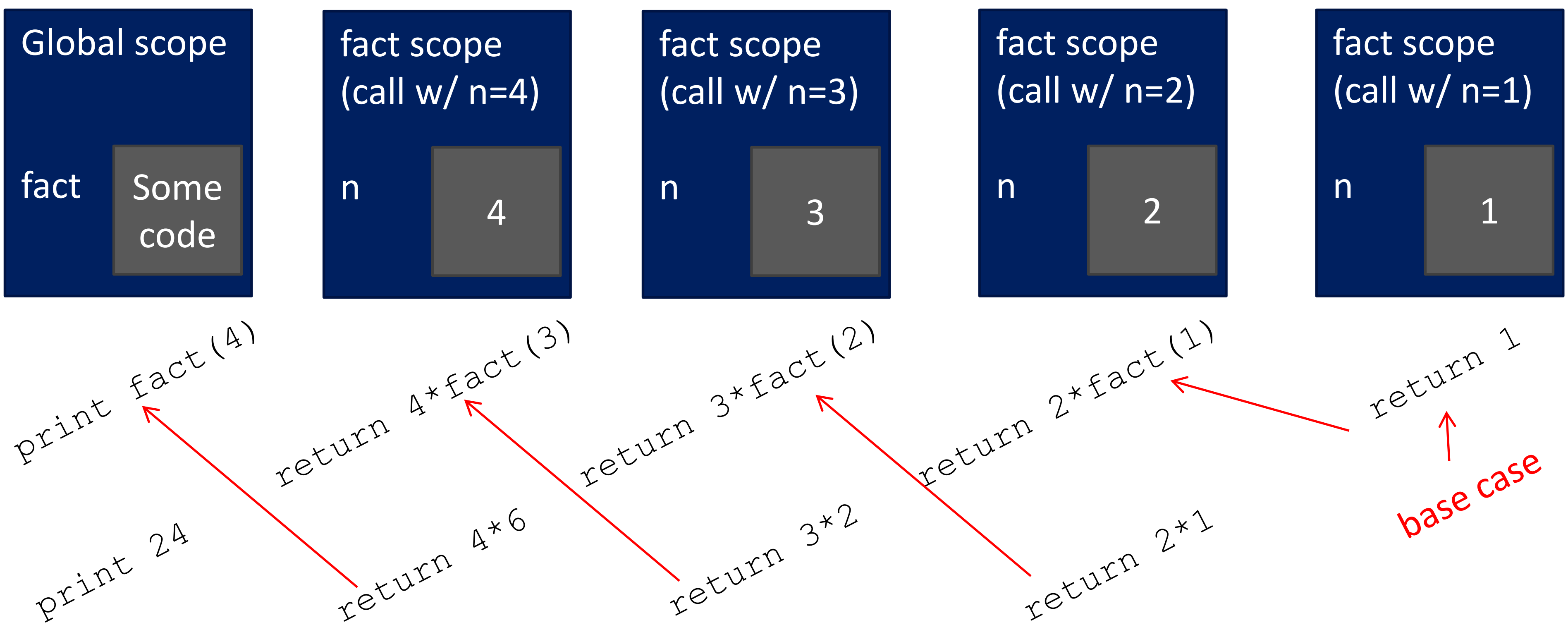
  n = 1 →
  ```
  if n == 1:
      return 1
  ```
  base case

- 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

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

print(fact(4))
```



SOME OBSERVATIONS

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

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

ITERATION vs. RECURSION

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

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

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

INDUCTIVE REASONING

- How do we know that our recursive code will work?
- `mult_iter` terminates because b is initially positive, and decreases by 1 each time around loop; thus must eventually become less than 1
- `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; must eventually reach call with b = 1

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


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

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 an arbitrary value of n, one can show that it must be true for n+1

EXAMPLE OF INDUCTION

- 0 + 1 + 2 + 3 + … + 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 some k, then need to show that
    - 0 + 1 + 2 + … + k + (k+1) = ((k+1)(k+2))/2
    - LHS is k(k+1)/2 + (k+1) by assumption that property holds for problem of size k
    - This becomes, by algebra, ((k+1)(k+2))/2
  - Hence expression holds for all n >= 0

RELEVANCE TO CODE?

- Same logic applies

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

- Base case, we can show that `mult` must return correct answer
- For recursive case, we can assume that `mult` correctly returns an answer for problems of size smaller than b, then by the addition step, it must also return a correct answer for problem of size b
- Thus by induction, code correctly returns answer

TOWERS OF HANOI

- The story:
  - 3 tall spikes
  - Stack of 64 different sized discs – start on one spike
  - Need to move stack to second spike (at which point universe ends)
  - Can only move one disc at a time, and a larger disc can never cover up a small disc

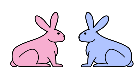
TOWERS OF HANOI

- 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

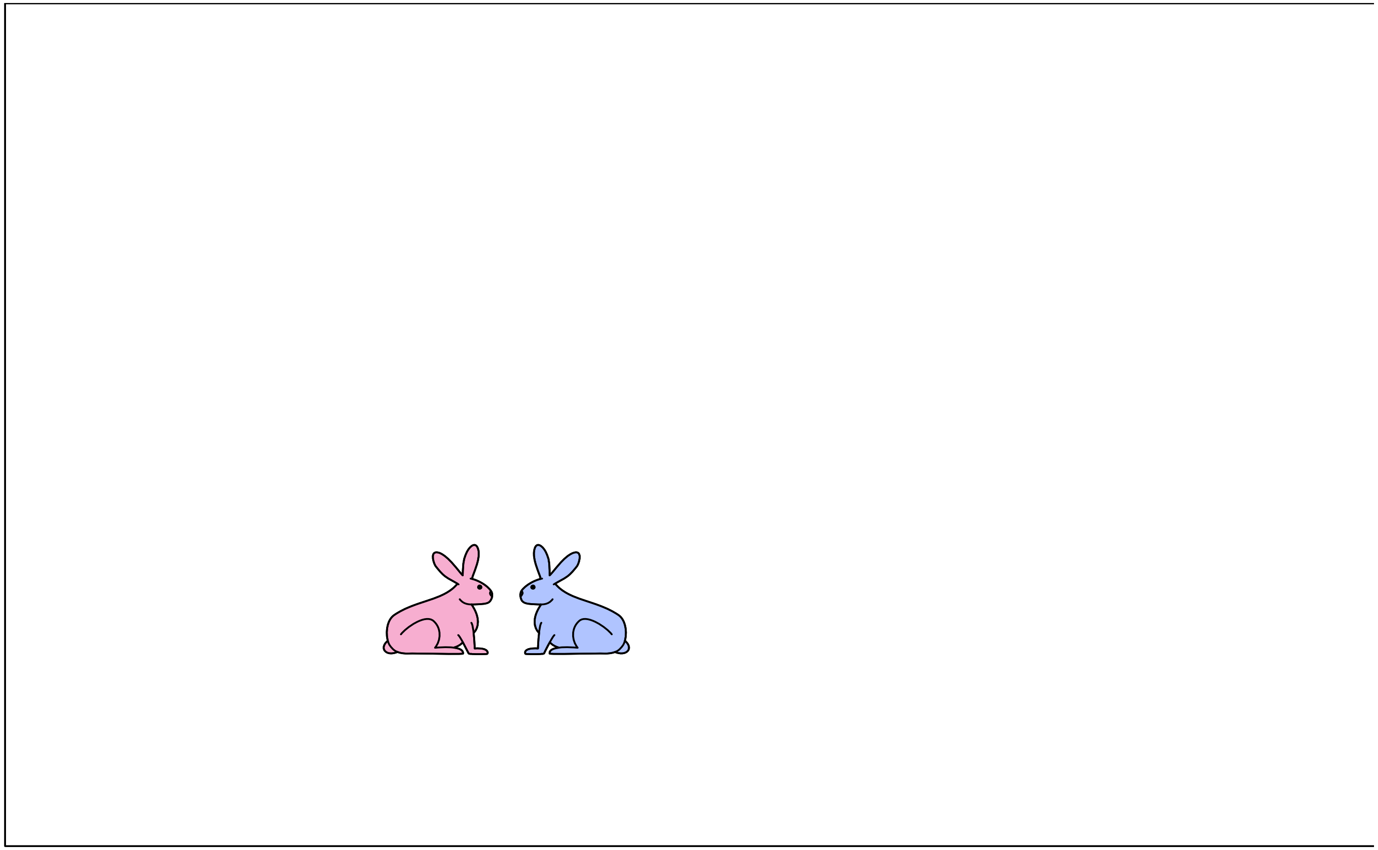
```python
def printMove(fr, to):
    print('move from ' + str(fr) + ' to ' + str(to))

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)
```

RECURSION WITH MULTIPLE BASE CASES

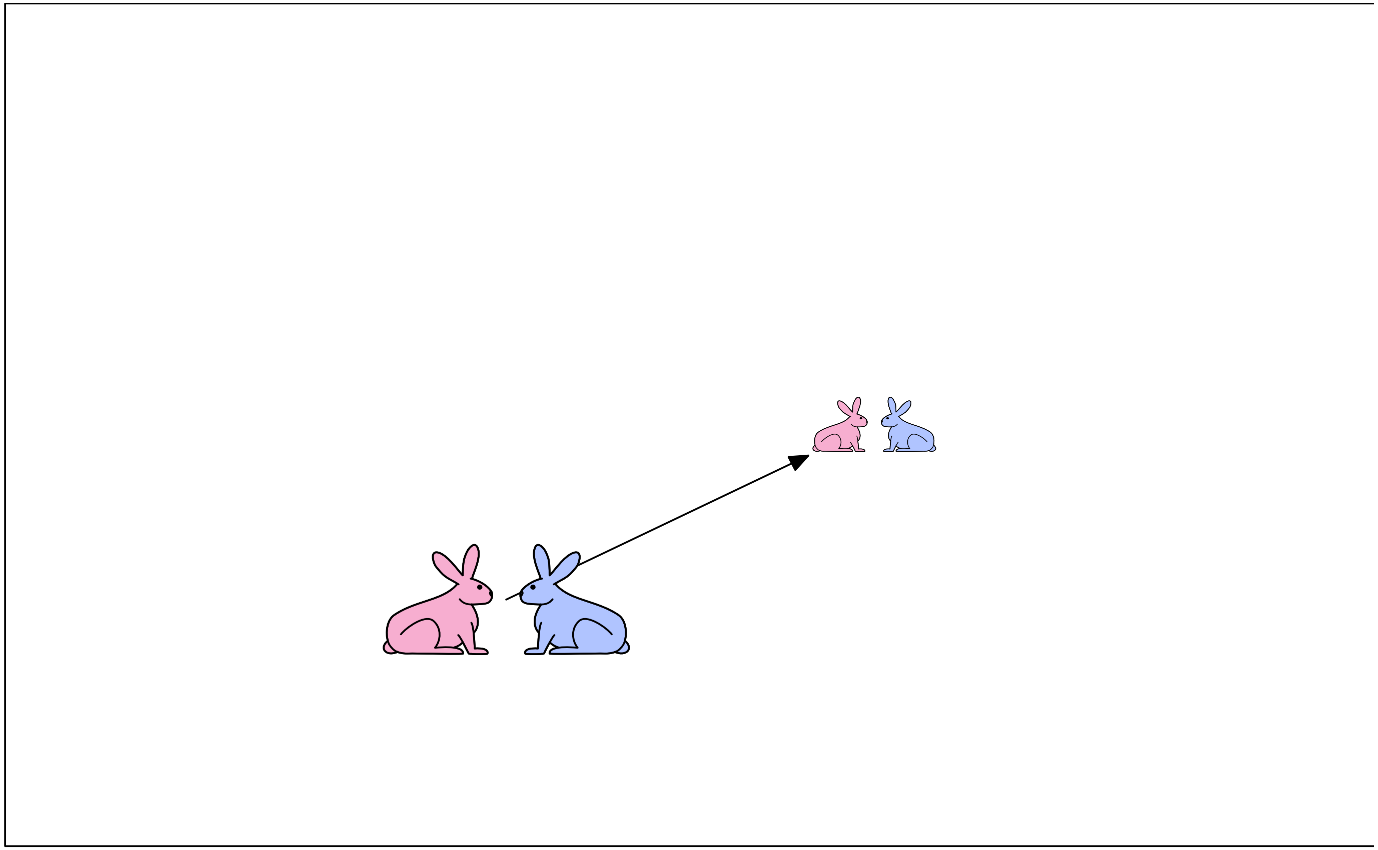
- Fibonacci numbers
  - Leonardo of Pisa (aka Fibonacci) modeled the following challenge
    - Newborn pair of rabbits (one female, one male) are put in a pen
    - Rabbits mate at age of one month
    - Rabbits have a one month gestation period
    - Assume rabbits never die, that female always produces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



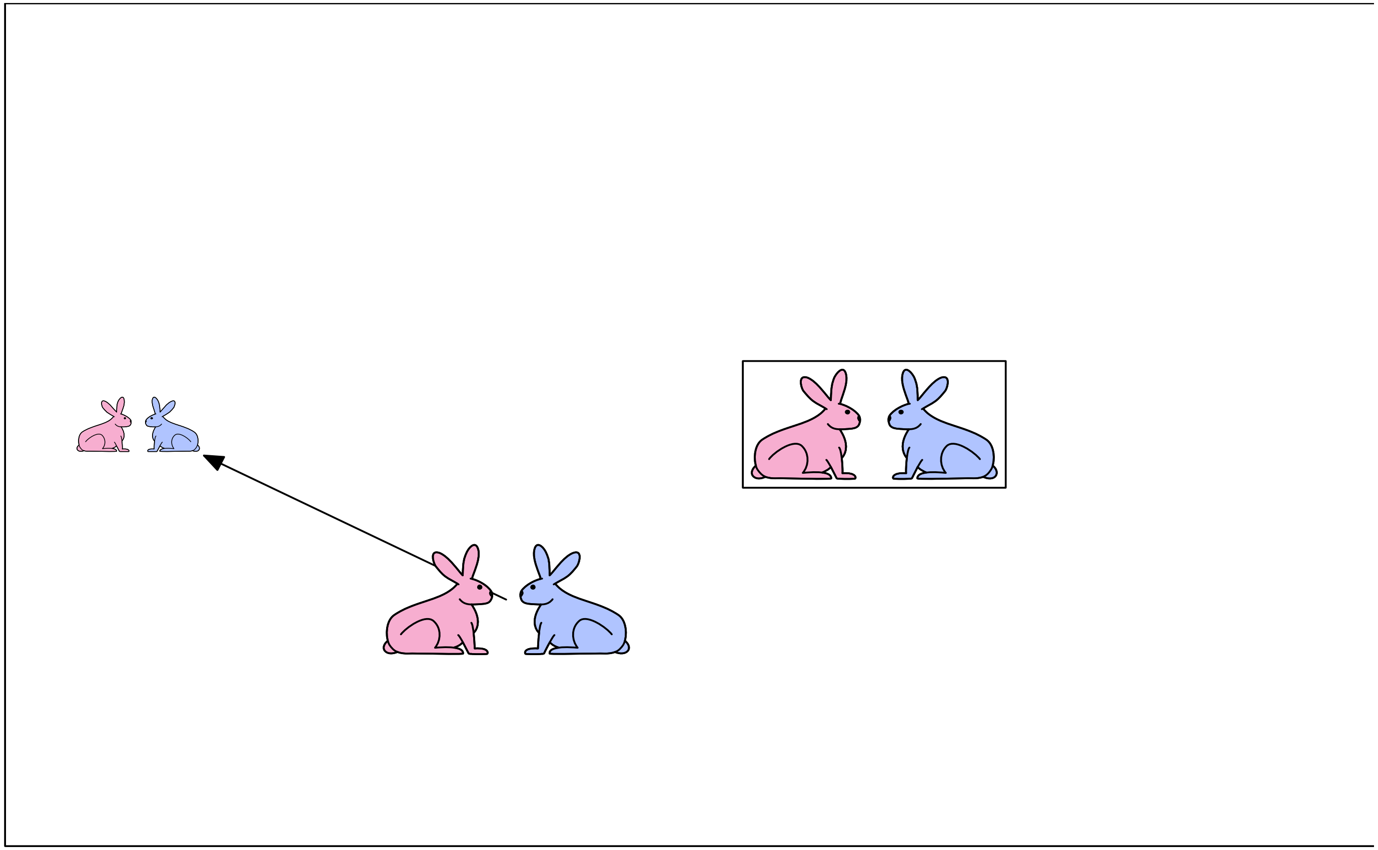
Demo courtesy of Prof. Denny Freeman and Adam Hartz



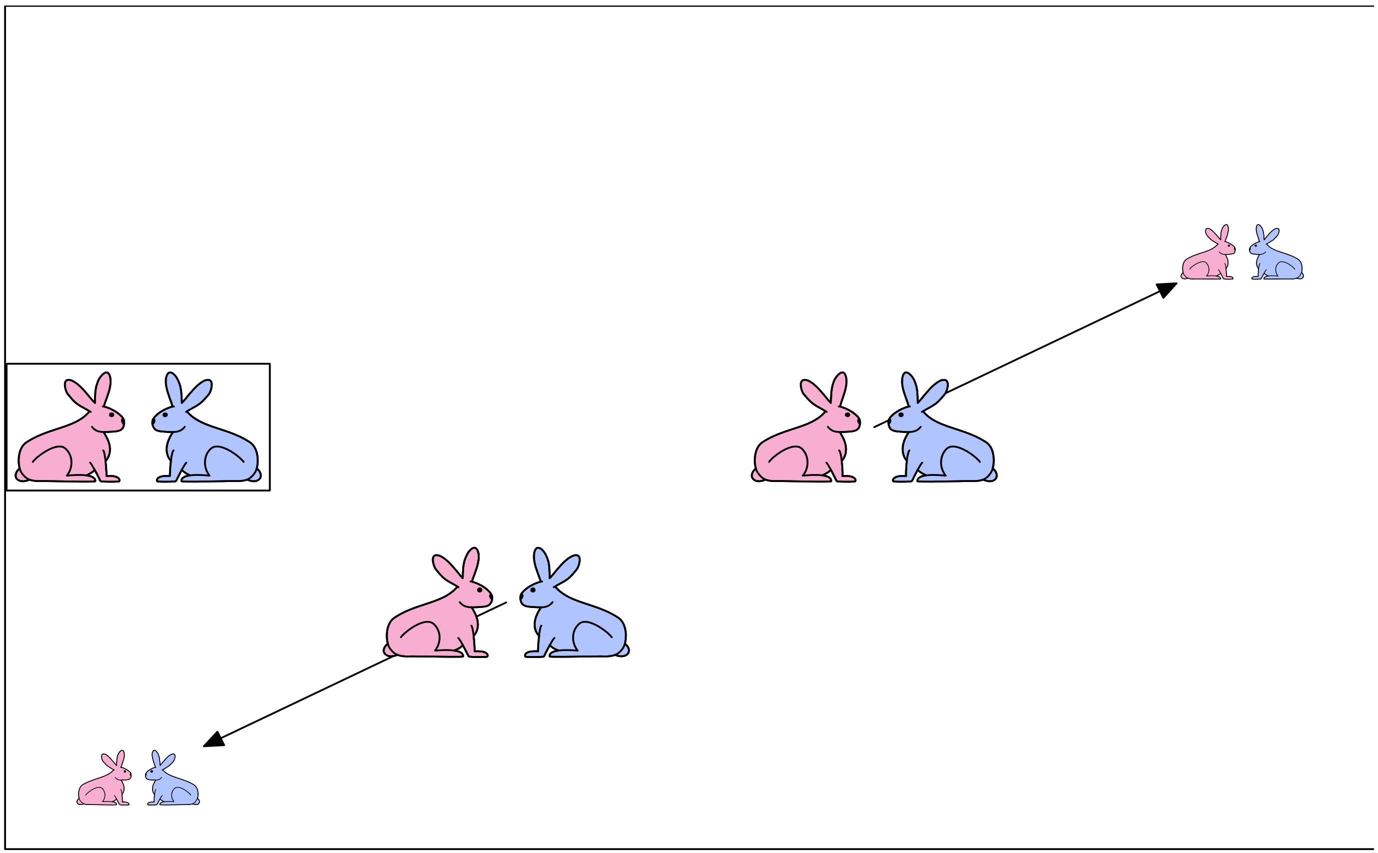
Demo courtesy of Prof. Denny Freeman and Adam Hartz



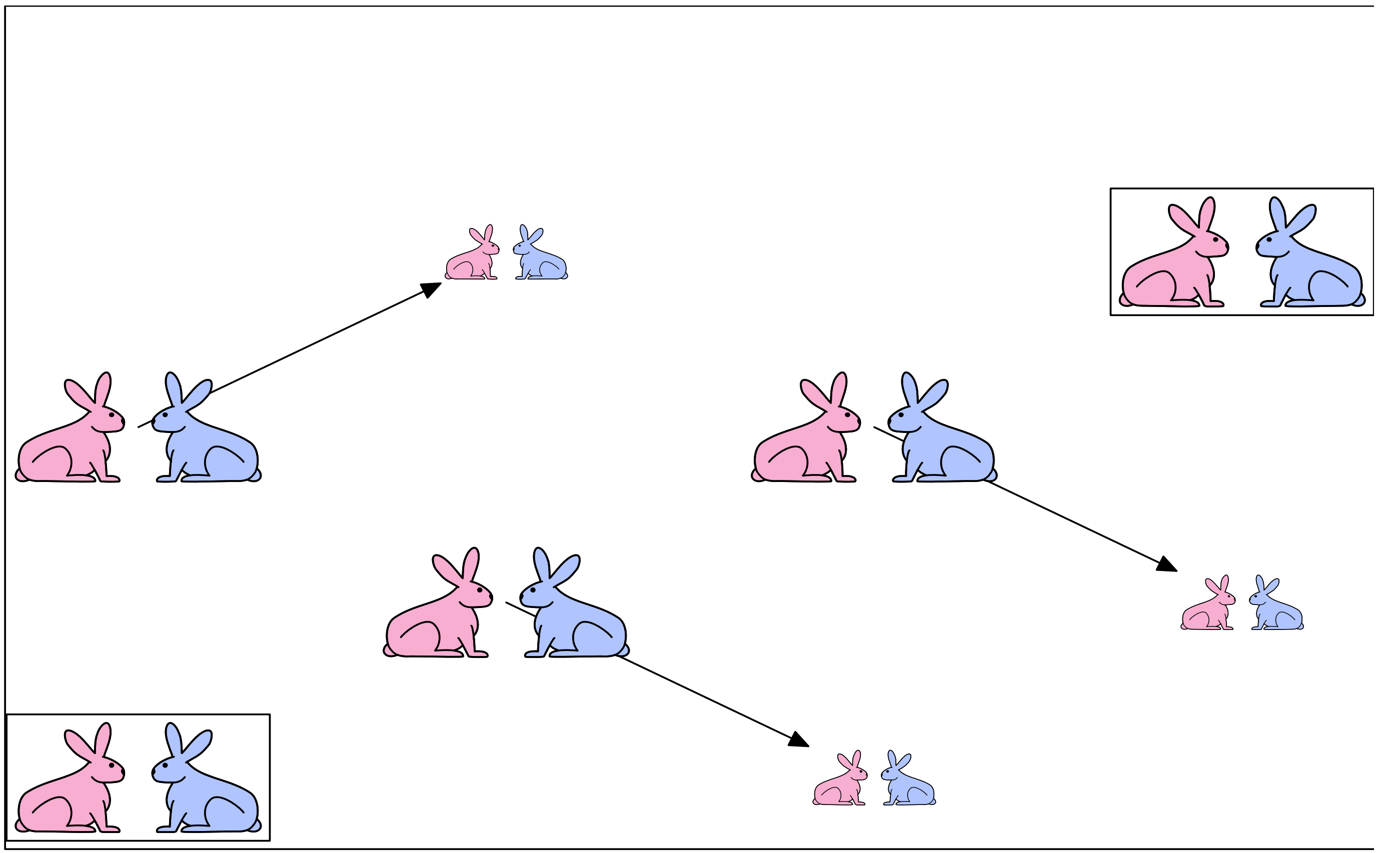
Demo courtesy of Prof. Denny Freeman and Adam Hartz



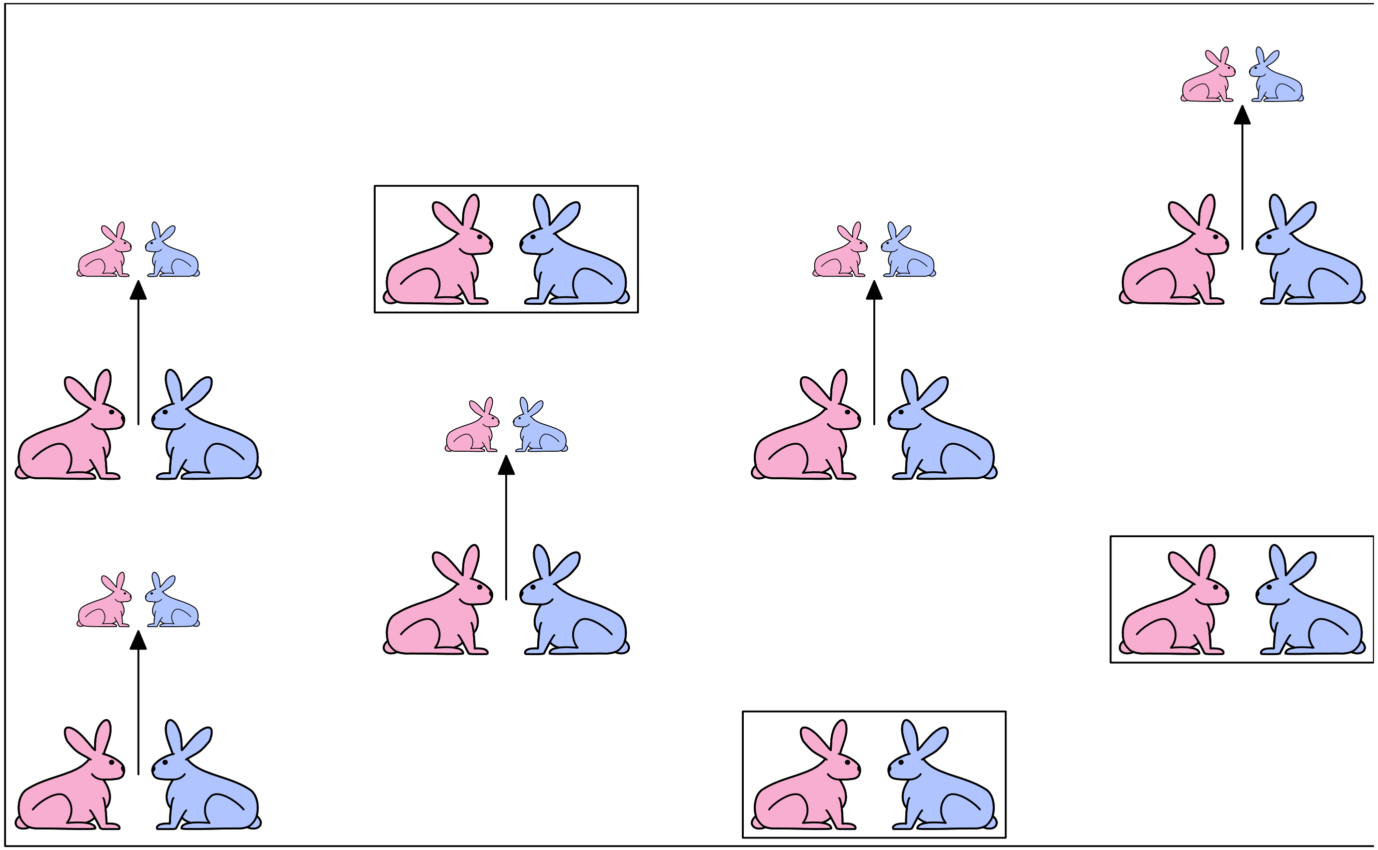
Demo courtesy of Prof. Denny Freeman and Adam Hartz



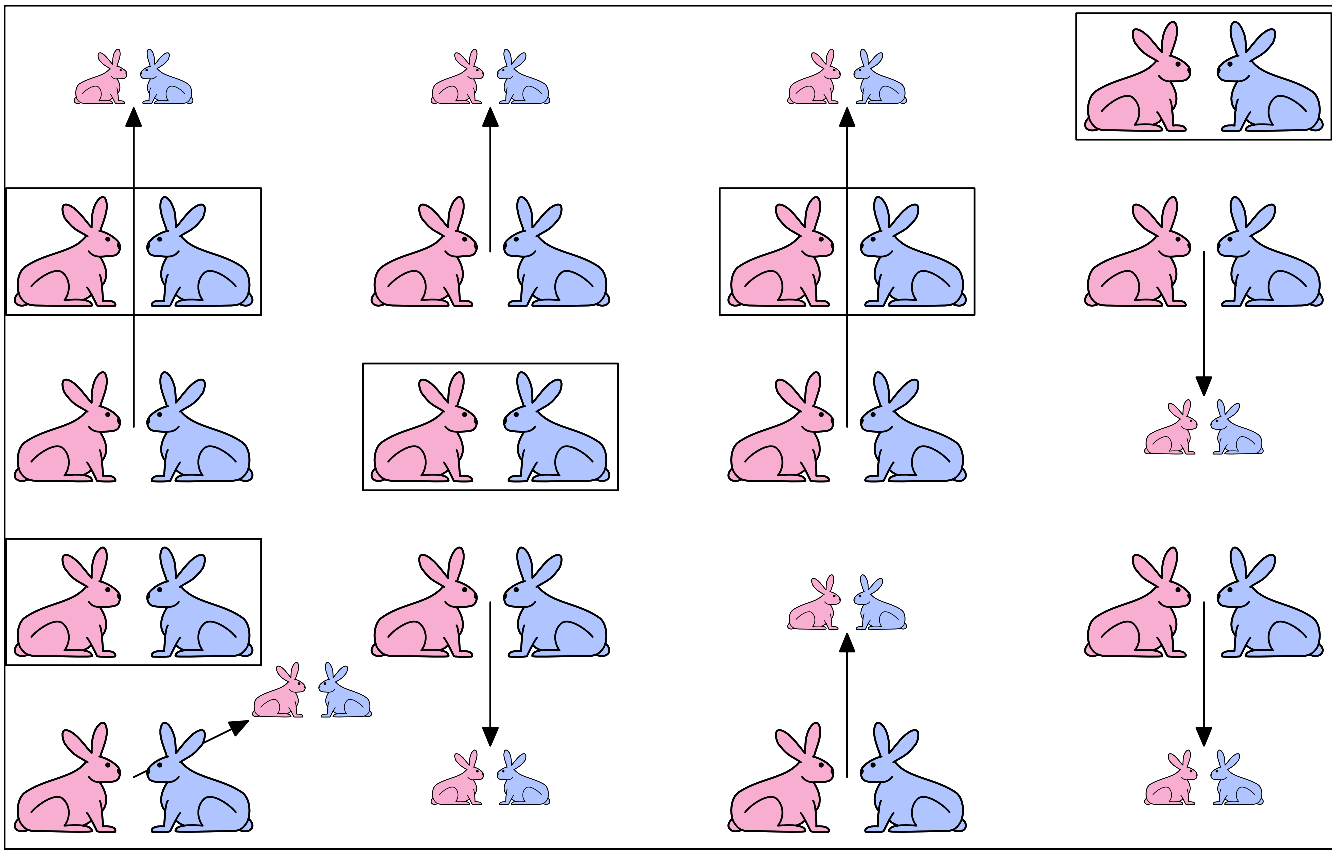
Demo courtesy of Prof. Denny Freeman and Adam Hartz

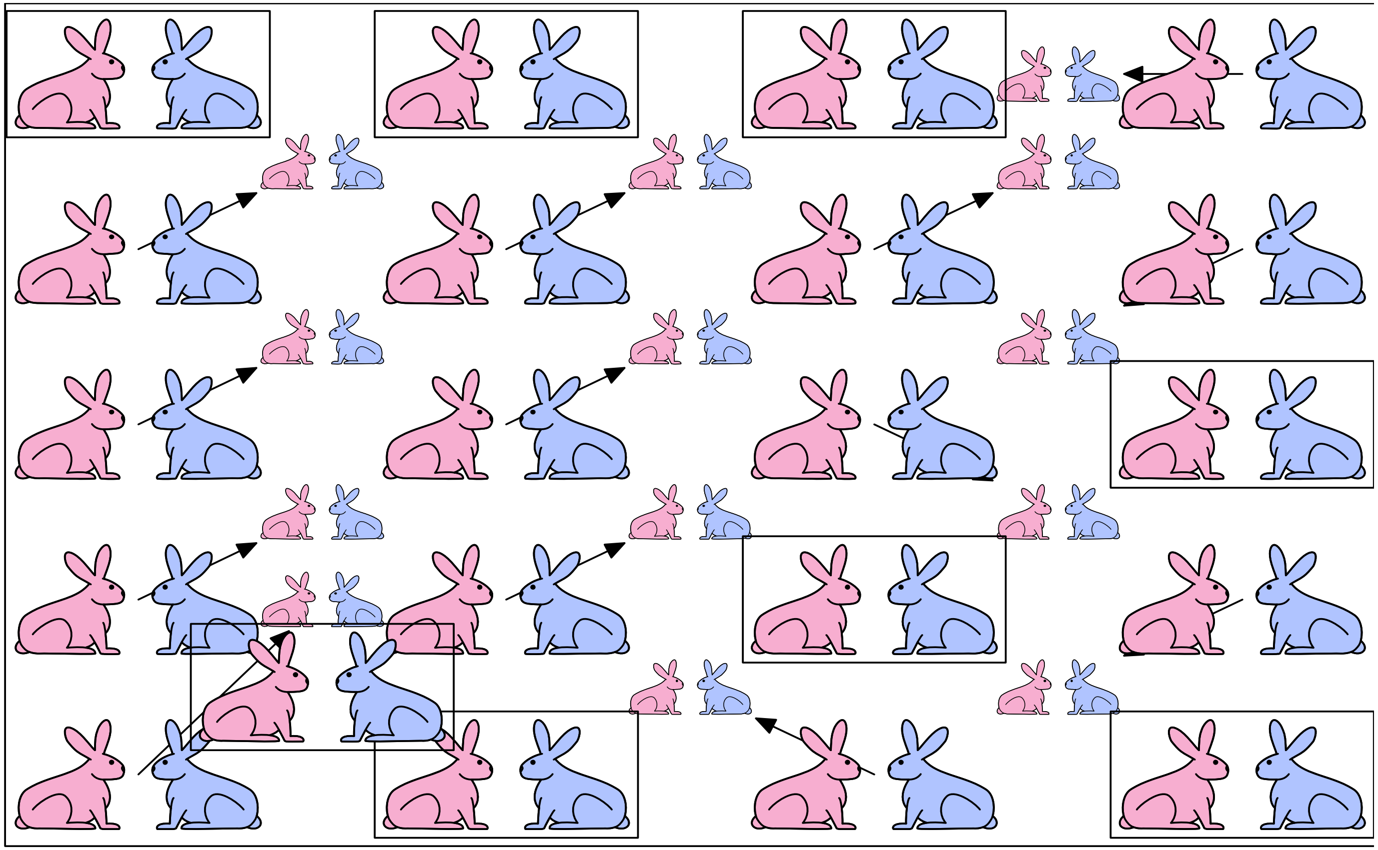


Demo courtesy of Prof. Denny Freeman and Adam Hartz

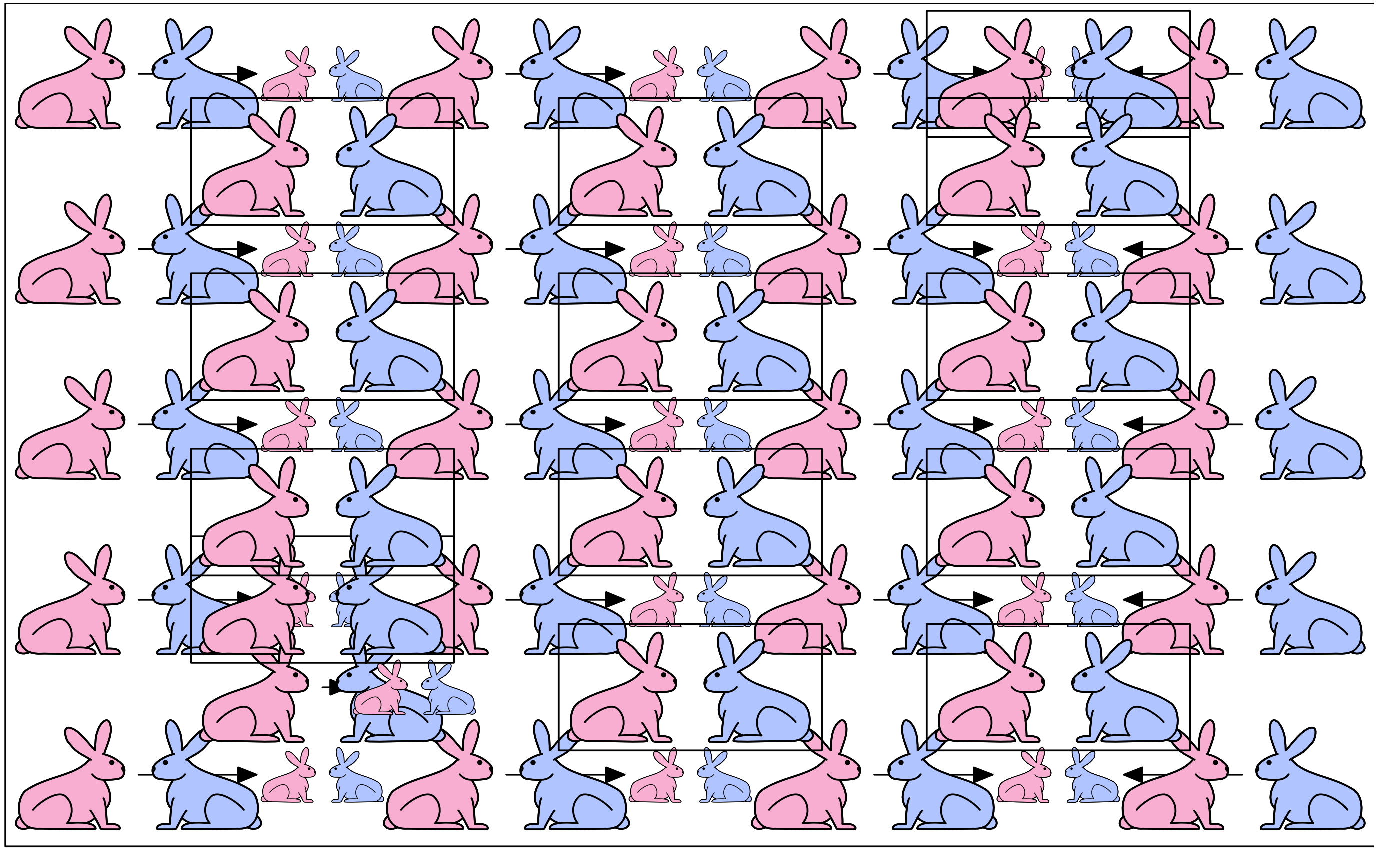


Demo courtesy of Prof. Denny Freeman and Adam Hartz





Demo courtesy of Prof. Denny Freeman and Adam Hartz



Demo courtesy of Prof. Denny Freeman and Adam Hartz

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
1	1
2	2
3	3
4	5
5	8
6	13

FIBONACCI

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

```
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)
```

RECURSION ON NON-NUMERICS

- how to check if a string of characters is a palindrome, i.e., reads the same forwards and backwards
  - “Able was I, ere I saw Elba” – attributed to Napoleon
  - “Are we not drawn onward, we few, drawn onward to new era?” – attributed to Anne Michaels

By Beinecke Library (Flickr: [General Napolean Bonaparte]) [CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0)], via Wikimedia Commons

By Larth_Rasnal (Own work) [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC BY 3.0 (http://creativecommons.org/licenses/by/3.0)], via Wikimedia Commons

SOLVING RECURSIVELY?

- First, convert the string to just characters, by stripping out punctuation, and converting upper case to lower case
- Then
  - Base case: a string of length 0 or 1 is a palindrome
  - Recursive case:
    - If first character matches last character, then is a palindrome if middle section is a palindrome

EXAMPLE

- ‘Able was I, ere I saw Elba’ → ‘ablewasiereisawleba’
- `isPalindrome('ablewasiereisawleba')` is same as
  - `'a' == 'a'` and `isPalindrome('blewasiereisawleb')`

```python
def isPalindrome(s):

    def toChars(s):
        s = s.lower()
        ans = ''
        for c in s:
            if c in 'abcdefghijklmnopqrstuvwxyz':
                ans = ans + c
        return ans

    def isPal(s):
        if len(s) <= 1:
            return True
        else:
            return s[0] == s[-1] and isPal(s[1:-1])

    return isPal(toChars(s))
```

DIVIDE AND CONQUER

- an example of a "divide and conquer" algorithm
- solve a hard problem by breaking it into a set of sub-problems such that:
  - sub-problems are easier to solve than the original
  - solutions of the sub-problems can be combined to solve the original

MODULES AND FILES

- have assumed that all our code is stored in one file
- cumbersome for large collections of code, or for code that should be used by many different other pieces of programming
- a **module** is a `.py` file containing a collection Python definitions and statements

EXAMPLE MODULE

- the file `circle.py` contains

```
pi = 3.14159
def area(radius):
    return pi*(radius**2)
def circumference(radius):
    return 2*pi*radius
```

-

EXAMPLE MODULE

- then we can import and use this module:

```
import circle
pi = 3
print(pi)
print(circle.pi)
print(circle.area(3))
print(circle.circumference(3))
```

- results in the following being printed:

```
3
3.14159
28.27431
18.849539999999998
```

OTHER IMPORTING

- if we don't want to refer to functions and variables by their module, and the names don't collide with other bindings, then we can use:

```
from circle import *
print(pi)
print(area(3))
```

- this has the effect of creating bindings within the current scope for all objects defined within `circle`
- statements within a module are executed only the first time a module is imported

FILES

- need a way to save our work for later use
- every operating system has its own way of handling files; Python provides an operating-system independent means to access files, using a **file handle**

```
nameHandle = open('kids', 'w')
```

- creates a file named `kids` and returns file handle which we can name and thus reference. The `w` indicates that the file is to opened for writing into.

FILES: example

```
nameHandle = open('kids', 'w')
for i in range(2):
    name = input('Enter name: ')
    nameHandle.write(name + '\')
nameHandle.close()
```

FILES: example

```
nameHandle = open('kids', 'r')
for line in nameHandle:
    print(line)
nameHandle.close()
```