Lecture 5
Tuples

Dr. Muhammad Jawad Khan

Robotics and Intelligent Machine Engineering Department,
School of Mechanical and Manufacturing,
National University of Sciences and Technology,
Islamabad, Pakistan.

\

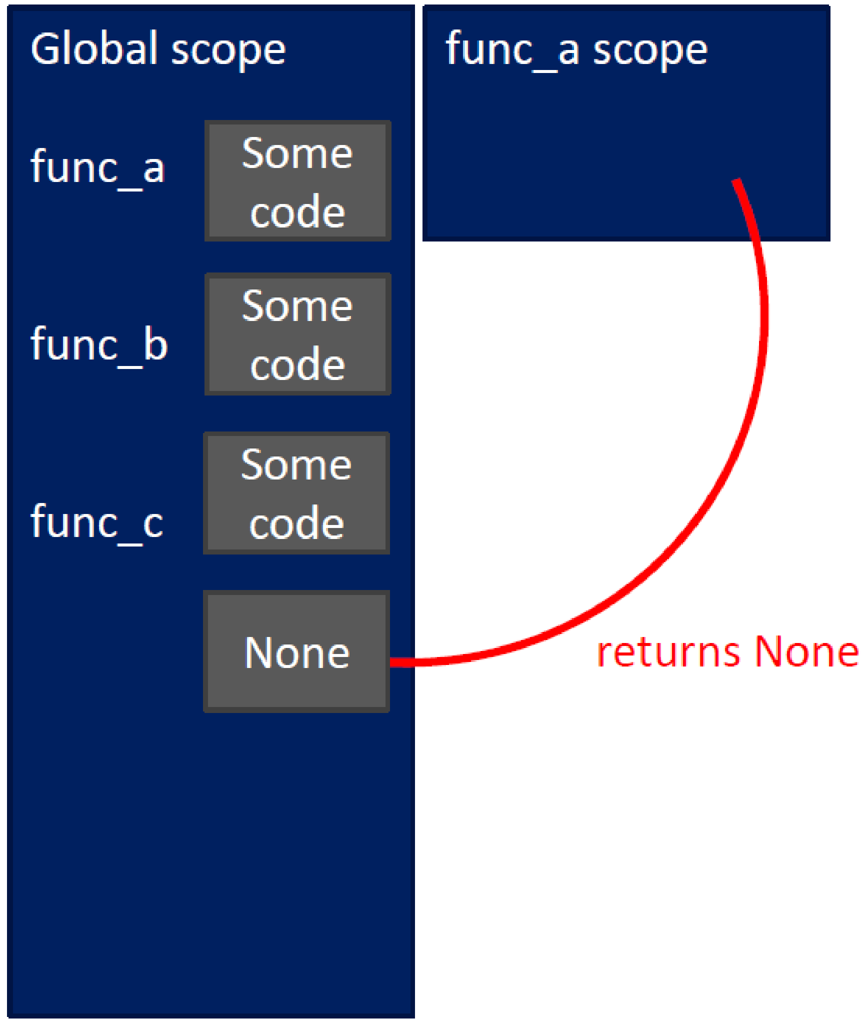
FUNCTIONS AS ARGUMENTS

- 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))
```

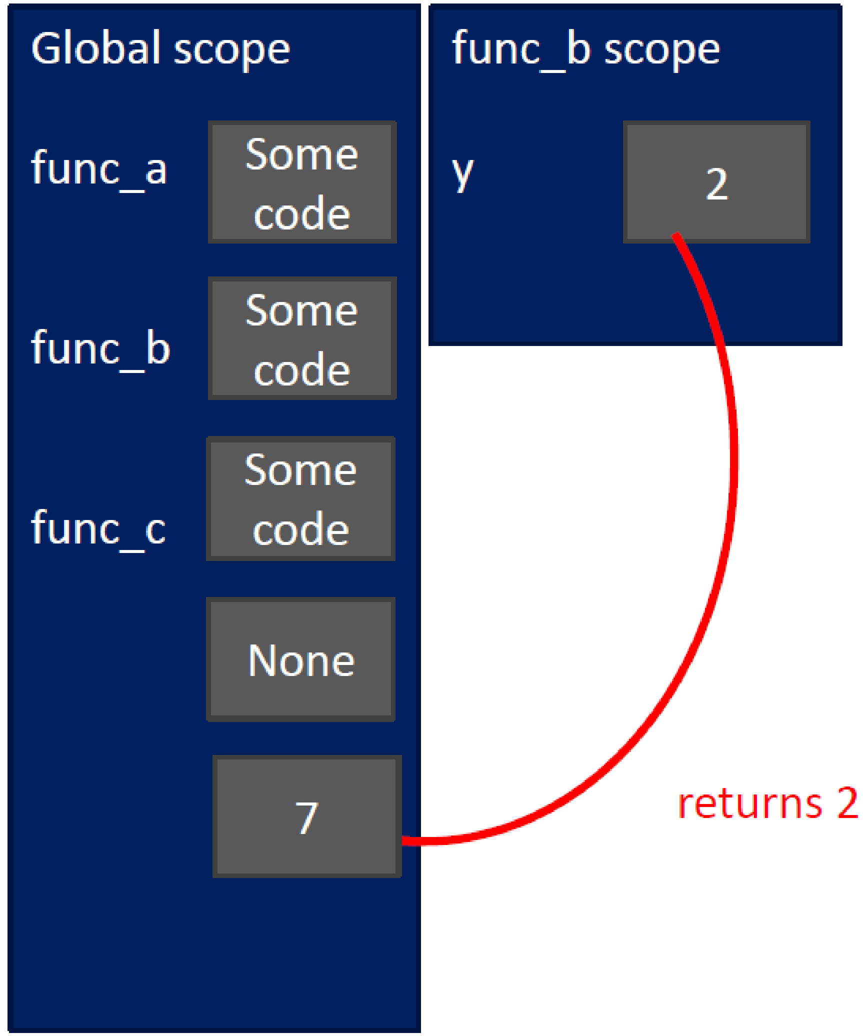
FUNCTIONS AS ARGUMENTS

```
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)
```



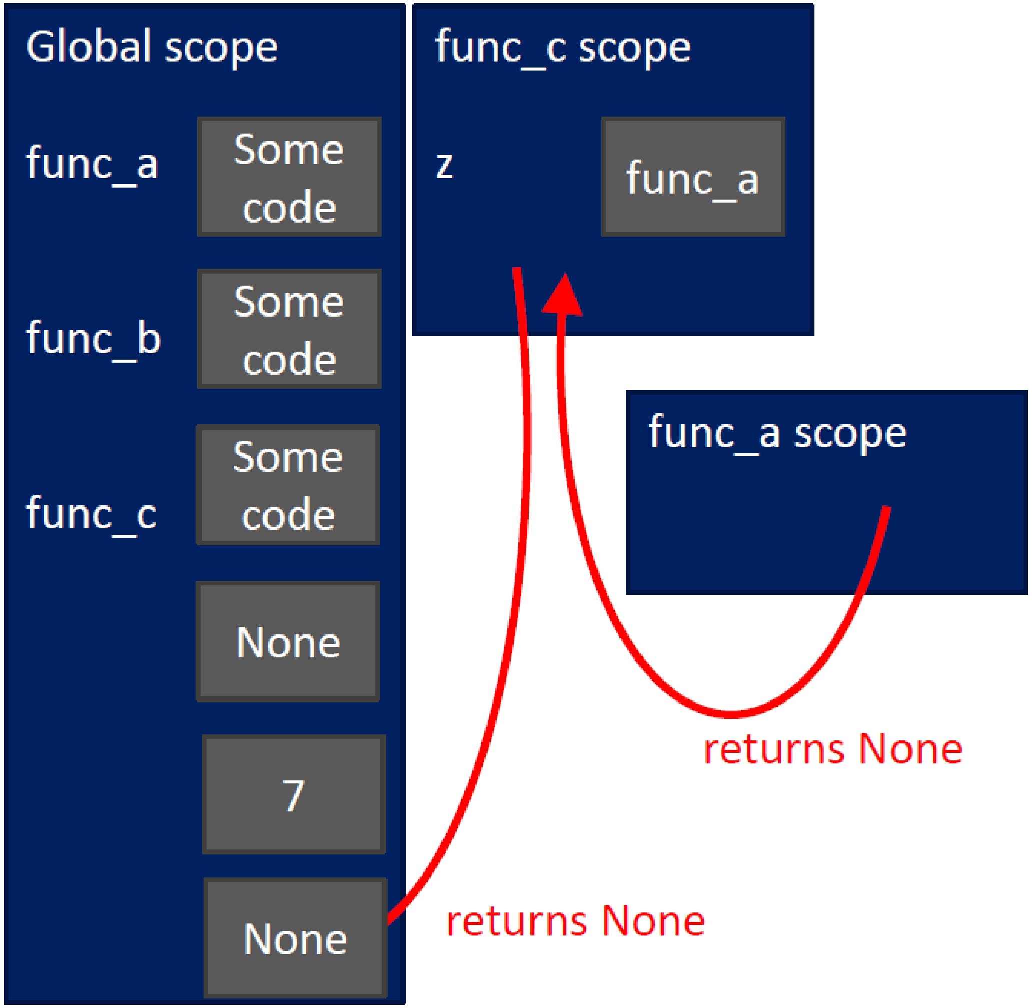
FUNCTIONS AS ARGUMENTS

```
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)
```



FUNCTIONS AS ARGUMENTS

```
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)
```



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

```python
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

```python
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

```python
def h(y):
    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 -- 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)
    print(x+1)
x = 5
g(x)
print(x)
```

```
def h(y):
    pass
    #x += 1 #leads to an error without
# line `global x` inside h
x = 5
h(x)
print(x)
```

HARDER SCOPE EXAMPLE

IMPORTANT
and
TRICKY!

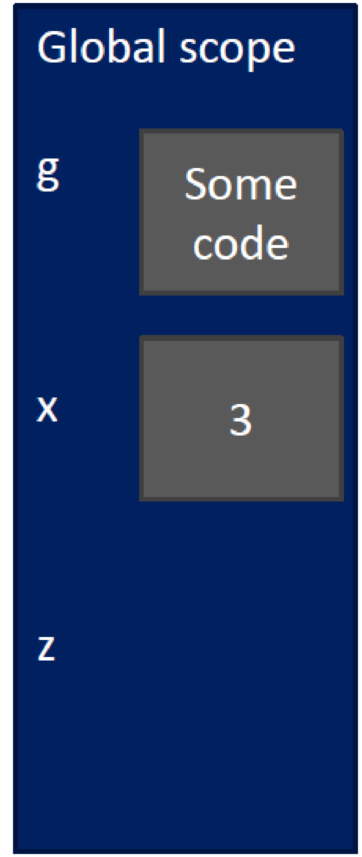
Python Tutor is your best friend to help sort this out!

http://www.pythontutor.com/

SCOPE DETAILS

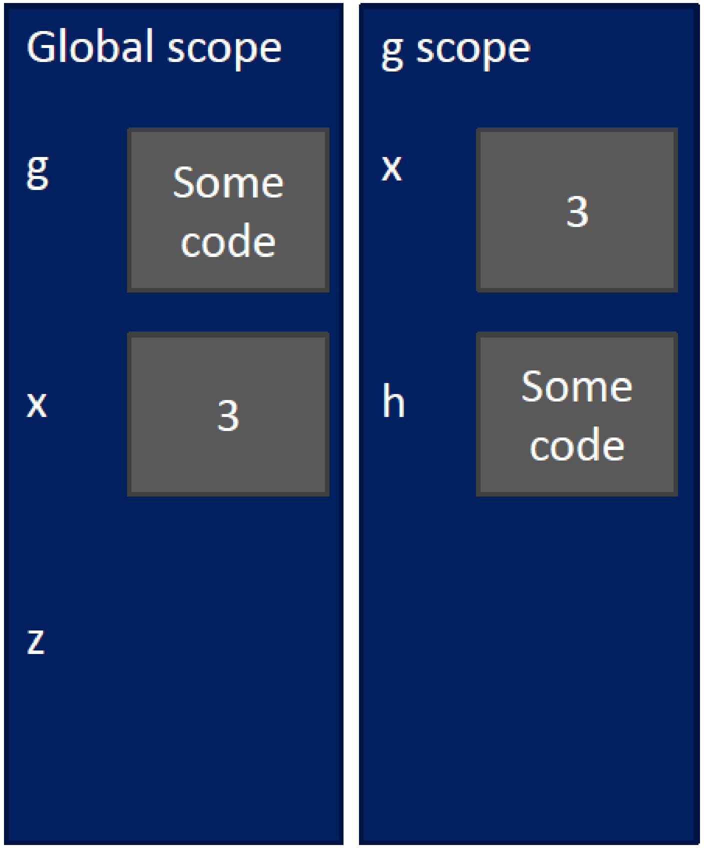
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

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

Some code



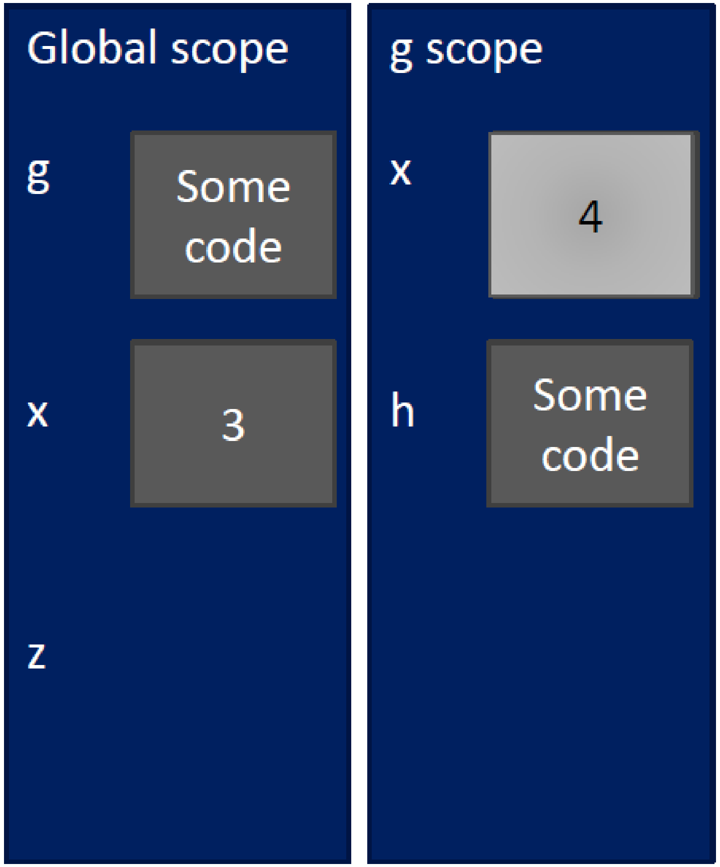
SCOPE DETAILS

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

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



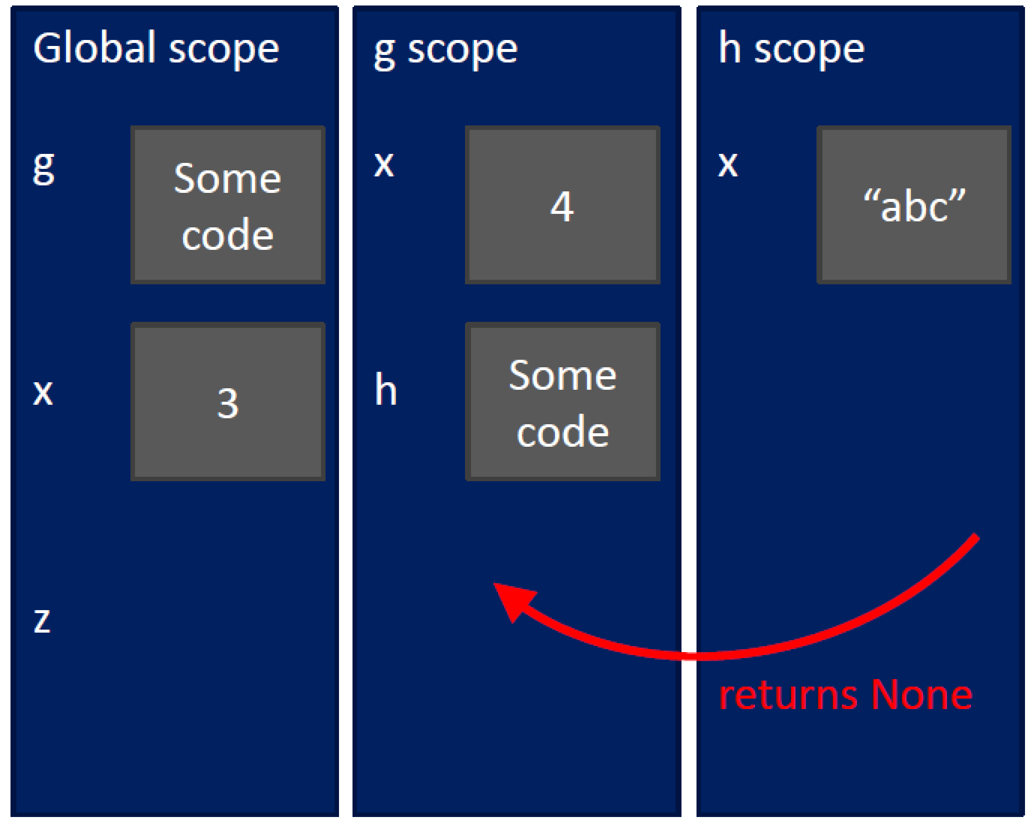
SCOPE DETAILS

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

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



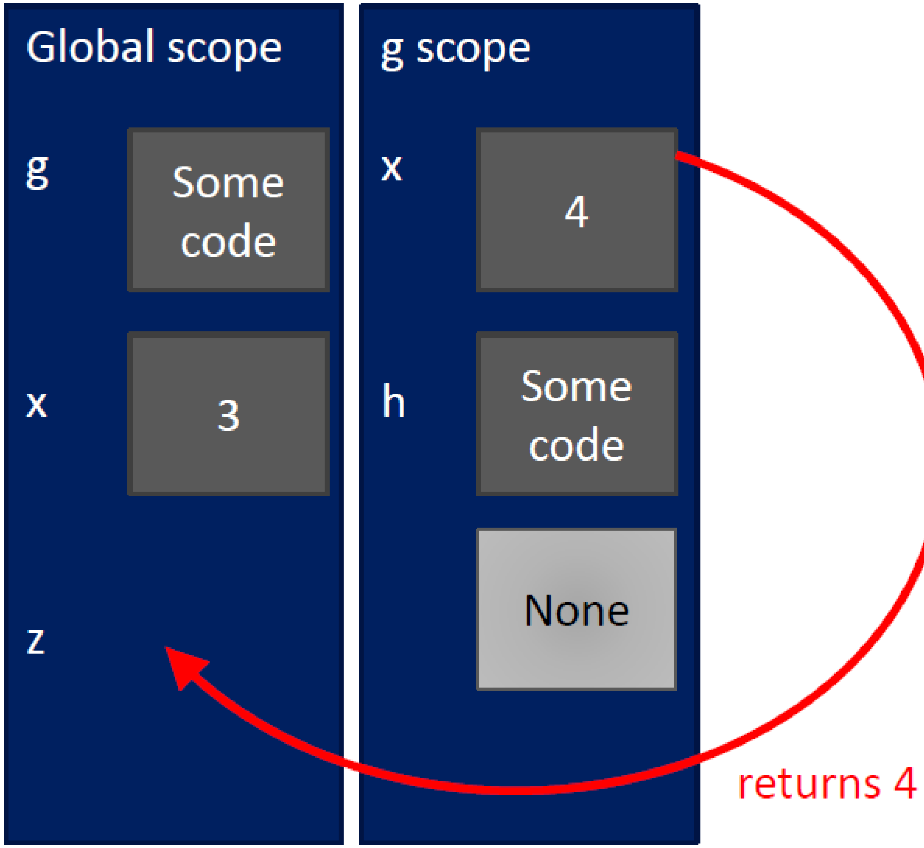
SCOPE DETAILS

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

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



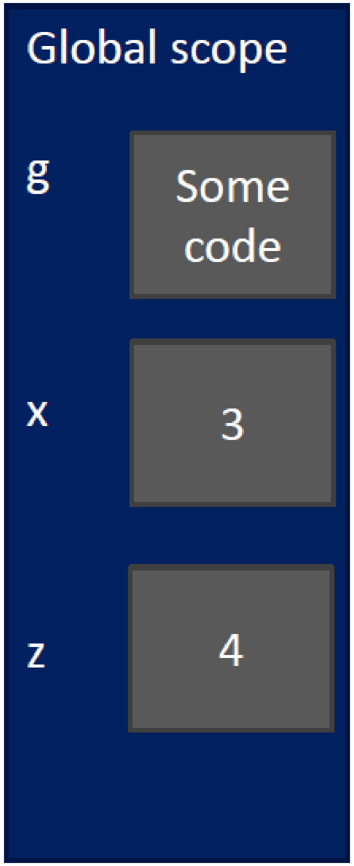
SCOPE DETAILS

```
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!

LAST TIME

- functions
- decomposition – create structure
- abstraction – suppress details
- from now on will be using functions a lot

TODAY

- have seen variable types: `int`, `float`, `bool`, `string`
- introduce new **compound data types**
  - tuples
  - lists
- idea of aliasing
- idea of mutability
- idea of cloning

TUPLES

- an ordered sequence of elements, can mix element types
- cannot change element values, **immutable**
- represented with parentheses

remember strings?

```
te = ()
```

empty tuple

```
t = (2,"mit",3)
```

`t[0]` → evaluates to 2

`(2,"mit",3) + (5,6)` → evaluates to `(2,"mit",3,5,6)`

`t[1:2]` → slice tuple, evaluates to `("mit",)`

`t[1:3]` → slice tuple, evaluates to `("mit",3)`

`len(t)` → evaluates to 3

`t[1] = 4` → gives error, can't modify object

extra comma means a tuple with one element

TUPLES

- conveniently used to **swap** variable values

```
x = y
y = x
```

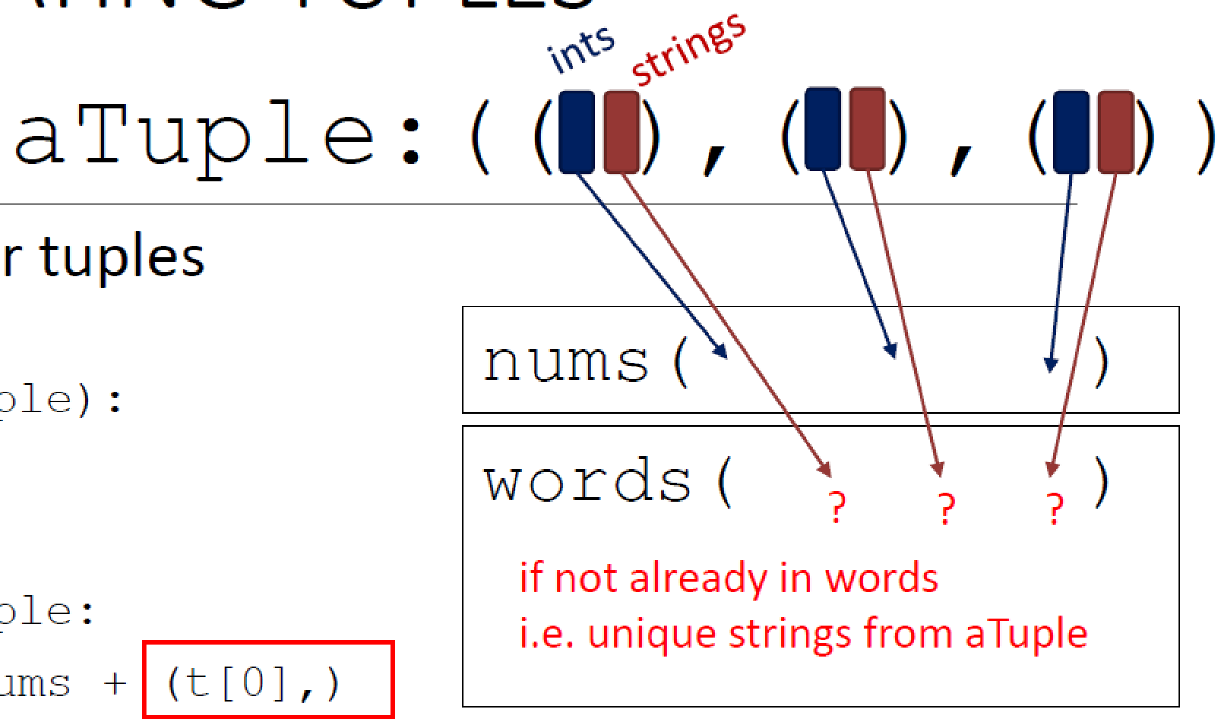
```
temp = x
x = y
y = temp
```

```
(x, y) = (y, x)
```

- used to **return more than one value** from a function

```
def quotient_and_remainder(x, y):
    q = x // y
    r = x % y
    return (q, r)

(quot, rem) = quotient_and_remainder(5,3)
print(quot)
print(rem)
```

MANIPULATING TUPLES



- can **iterate** over tuples

empty tuple

singleton tuple

```
def get_data(aTuple):
    nums = ()
    words = ()
    for t in aTuple:
        nums = nums + (t[0],)
        if t[1] not in words:
            words = words + (t[1],)
    min_n = min(nums)
    max_n = max(nums)
    unique_words = len(words)
    return (min_n, max_n, unique_words)
```

```
def get_data(aTuple):
nums = ()    # empty tuple
    words = ()
    for t in aTuple:
        # concatenating with a singleton tuple
        nums = nums + (t[0],)
        # only add words haven't added before
        if t[1] not in words:
            words = words + (t[1],)
    min_n = min(nums)
    max_n = max(nums)
    unique_words = len(words)
    return (min_n, max_n, unique_words)
```

LISTS

- **ordered sequence** of information, accessible by index
- a list is denoted by **square brackets**, [ ]
- a list contains **elements**
  - usually homogeneous (ie, all integers)
  - can contain mixed types (not common)
- list elements can be changed so a list is **mutable**

INDICES AND ORDERING

`a_list = []` empty list

`L = [2, 'a', 4, [1,2]]`

`len(L)` → evaluates to 4

`L[0]` → evaluates to 2

`L[2]+1` → evaluates to 5

`L[3]` → evaluates to `[1,2]`, another list!

`L[4]` → gives an error

```
i = 2
```

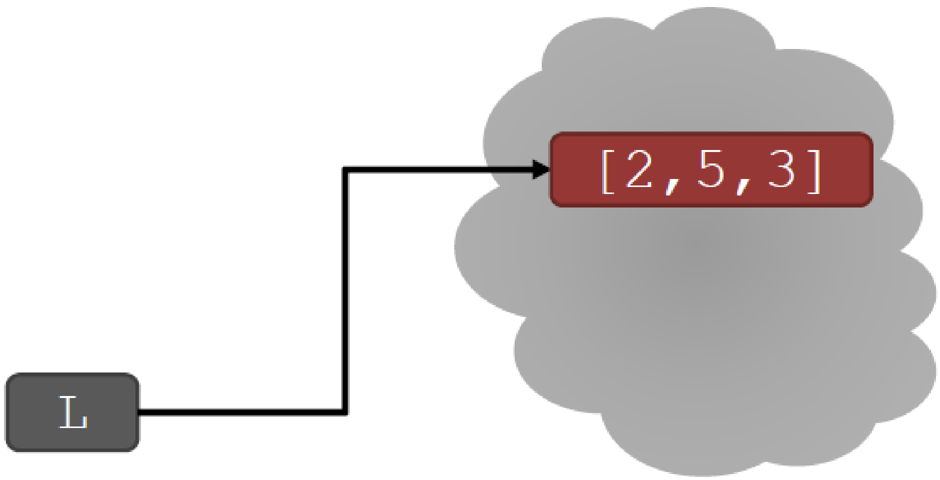
`L[i-1]` → evaluates to 'a' since `L[1]='a'` above

CHANGING ELEMENTS

- lists are **mutable**!
- assigning to an element at an index changes the value

```
L = [2, 1, 3]
L[1] = 5
```

- `L` is now `[2, 5, 3]`, note this is the **same object** `L`



ITERATING OVER A LIST

- compute the **sum of elements** of a list
- common pattern, iterate over list elements

```
total = 0
for i in range(len(L)):
    total += L[i]
print total
```

```
total = 0
for i in L:
    total += i
print total
```

like strings, can iterate over list elements directly

- notice
  - list elements are indexed `0` to `len(L)-1`
  - `range(n)` goes from `0` to `n-1`

```python
def sum_elem_method1(L):
  total = 0
  for i in range(len(L)):
      total += L[i]
  return total
```

```python
def sum_elem_method2(L):
    total = 0
    for i in L:
        total += i
    return total
```

```python
print(sum_elem_method1([1,2,3,4]))
print(sum_elem_method2([1,2,3,4]))
```

OPERATIONS ON LISTS - ADD

- **add** elements to end of list with `L.append(element)`
- **mutates** the list!

```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`

- what is the dot?
  - lists are Python objects, everything in Python is an object
  - objects have data
  - objects have methods and functions
  - access this information by `object_name.do_something()`
  - will learn more about these later

OPERATIONS ON LISTS - ADD

- to combine lists together use **concatenation**, + operator, to give you a new list
- **mutate** list with `L.extend(some_list)`

```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2        → L3 is [2,1,3,4,5,6]
                      L1, L2 unchanged

L1.extend([0,6])    → mutated L1 to [2,1,3,0,6]
```

OPERATIONS ON LISTS - REMOVE

- delete element at a **specific index** with `del(L[index])`
- remove element at **end of list** with `L.pop()`, returns the removed element
- remove a **specific element** with `L.remove(element)`
  - looks for the element and removes it
  - if element occurs multiple times, removes first occurrence
  - if element not in list, gives an error

all these operations mutate the list

```
L = [2,1,3,6,3,7,0]    # Do below in order
L.remove(2)            → mutates L = [1,3,6,3,7,0]
L.remove(3)            → mutates L = [1,6,3,7,0]
del(L[1])              → mutates L = [1,3,7,0]
print(L.pop())         → returns 0 and mutates L = [1,3,7]
```

CONVERT LISTS TO STRINGS AND BACK

- convert **string to list** with `list(s)`, returns a list with every character from `s` an element in `L`
- can use `s.split()`, to **split a string on a character** parameter, splits on spaces if called without a parameter
- use `''.join(L)` to turn a **list of characters into a string**, can give a character in quotes to add char between every element

`s = "I<3 cs"`	→ `s` is a string
`print(list(s))`	→ returns `['I','<','3',' ','c','s']`
`print(s.split('<'))`	→ returns `['I', '3 cs']`
`L = ['a', 'b', 'c']`	→ `L` is a list
`print(''.join(L))`	→ returns `"abc"`
`print('_'.join(L))`	→ returns `"a_b_c"`

OTHER LIST OPERATIONS

- `sort()` and `sorted()`
- `reverse()`
- and many more!

https://docs.python.org/3/tutorial/datastructures.html

```
L=[9,6,0,3]
print(sorted(L))
L.sort()
L.reverse()
```

→ returns sorted list, does **not mutate** `L`

→ **mutates** `L=[0,3,6,9]`

→ **mutates** `L=[9,6,3,0]`

MUTATION, ALIASING, CLONING

IMPORTANT
and
TRICKY!

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

http://www.pythontutor.com/

LISTS IN MEMORY

- lists are **mutable**
- behave differently than immutable types
- is an object in memory
- variable name points to object
- any variable pointing to that object is affected
- key phrase to keep in mind when working with lists is **side effects**

AN ANALOGY

- attributes of a person
  - singer, rich
- he is known by many names
- all nicknames point to the **same person**
  - add new attribute to **one nickname** ...

Justin Bieber	singer	rich	troublemaker

  - ... **all his nicknames** refer to old attributes AND all new ones

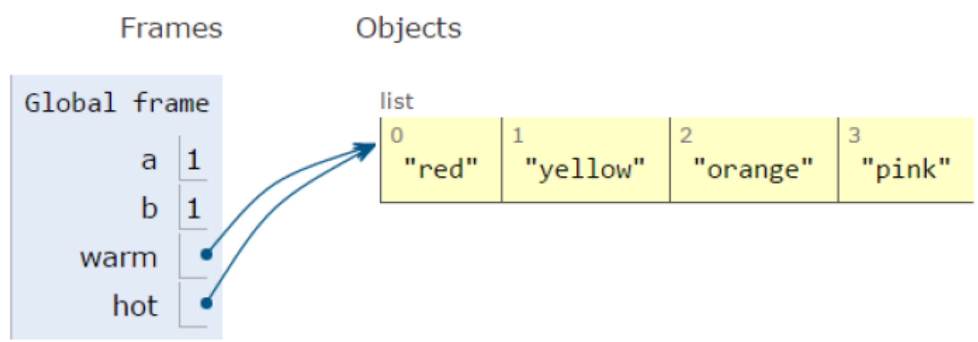
The Bieb	singer	rich	troublemaker
JBeebs	singer	rich	troublemaker

ALIASES

- `hot` is an **alias** for `warm` – changing one changes the other!
- `append()` has a side effect

```
a = 1
b = a
print(a)
print(b)

warm = ['red', 'yellow', 'orange']
hot = warm
hot.append('pink')
print(hot)
print(warm)
```

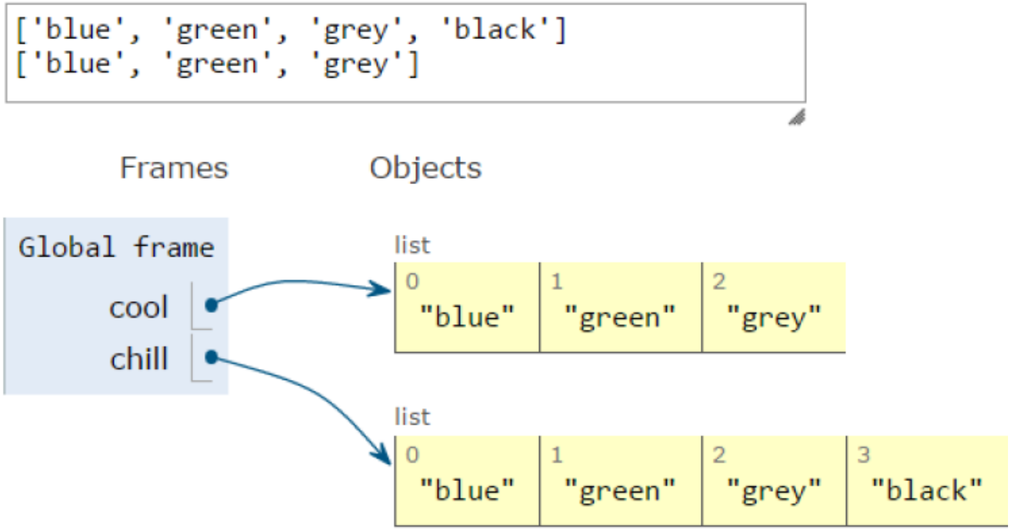
```
1
1
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```



CLONING A LIST

- create a new list and **copy every element** using
  `chill = cool[:]`

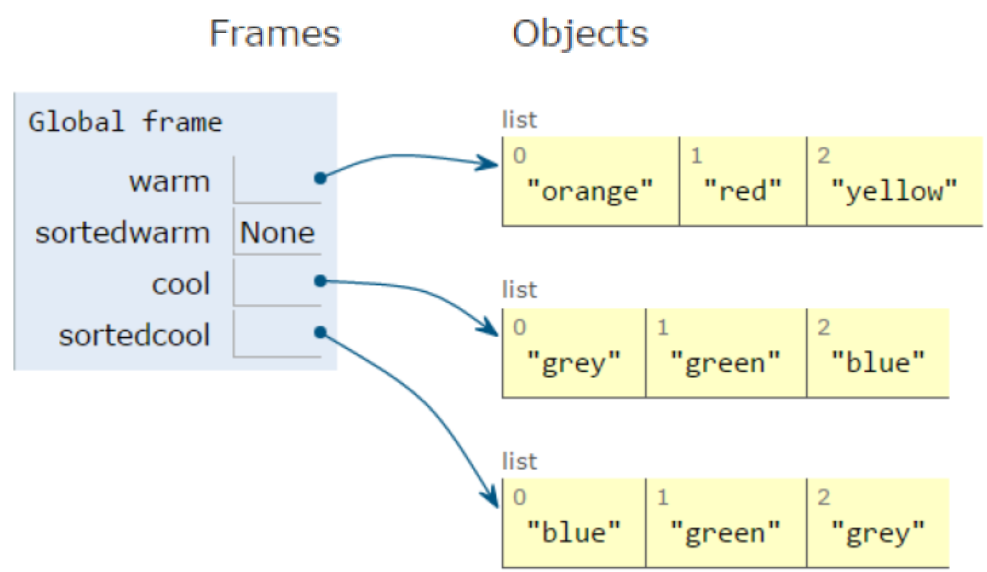
```
cool = ['blue', 'green', 'grey']
chill = cool[:]
chill.append('black')
print(chill)
print(cool)
```



SORTING LISTS

- calling `sort()` **mutates** the list, returns nothing
- calling `sorted()` **does not mutate** list, must assign result to a variable

```
['orange', 'red', 'yellow']
None
['grey', 'green', 'blue']
['blue', 'green', 'grey']
```



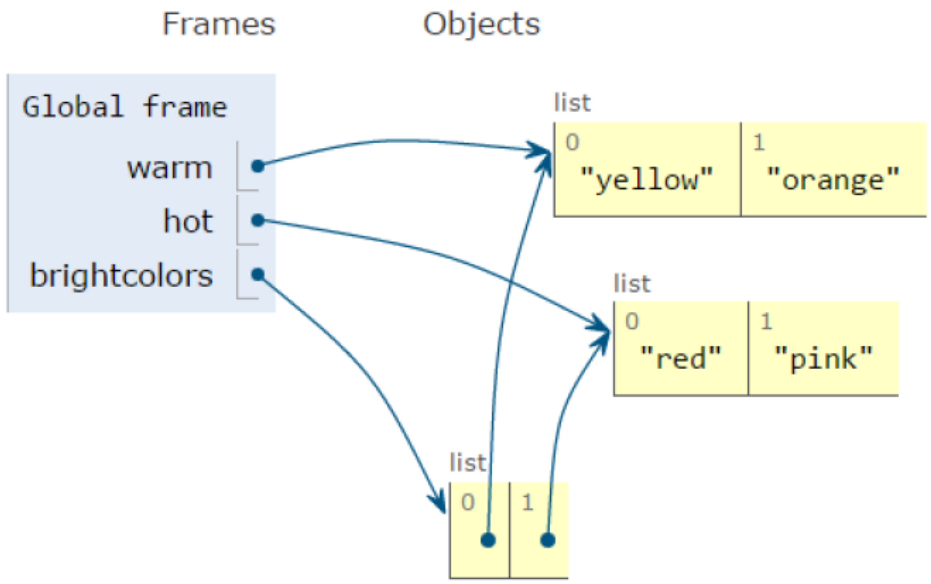

```
warm = ['red', 'yellow', 'orange']
sortedwarm = warm.sort()
print(warm)
print(sortedwarm)

cool = ['grey', 'green', 'blue']
sortedcool = sorted(cool)
print(cool)
print(sortedcool)
```

LISTS OF LISTS OF LISTS OF....

- can have **nested** lists
- side effects still possible after mutation

```
[['yellow', 'orange'], ['red']]
['red', 'pink']
[['yellow', 'orange'], ['red', 'pink']]
```

```
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
hot.append('pink')
print(hot)
print(brightcolors)
```



MUTATION AND ITERATION Try this in Python Tutor!

- **avoid** mutating a list as you are iterating over it

```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)
```

```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

```
L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
remove_dups(L1, L2)
```

clone list first, note that `L1_copy = L1` does NOT clone

- `L1` is `[2,3,4]` not `[3,4]` Why?
  - Python uses an internal counter to keep track of index it is in the loop
  - mutating changes the list length but Python doesn't update the counter
  - loop never sees element 2

```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)


L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
remove_dups(L1, L2)
print(L1, L2)
```

```
def remove_dups_new(L1,
L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
remove_dups_new(L1, L2)
print(L1, L2)
```

```python
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)



L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
remove_dups(L1, L2)
print(L1, L2)
```

```python
def remove_dups_new(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
remove_dups_new(L1, L2)
print(L1, L2)
```

Previously

- tuples - immutable
- lists - mutable
- aliasing, cloning
- mutability side effects

Now

- recursion – divide/decrease and conquer
- dictionaries – another mutable object type

RECURSION

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

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
- 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
    - must solve the same problem on **some other input** with the goal of simplifying the larger problem input

ITERATIVE ALGORITHMS SO FAR

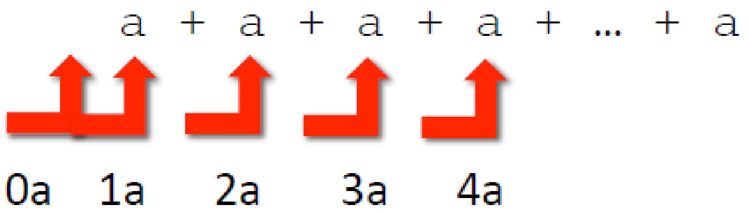
- looping constructs (`while` and `for` loops) lead to **iterative** algorithms
- can capture computation in a set of **state variables** that update on each iteration through loop

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

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

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

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

recursive reduction

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

base case

recursive step

FACTORIAL

`n! = n*(n-1)*(n-2)*(n-3)* … * 1`

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

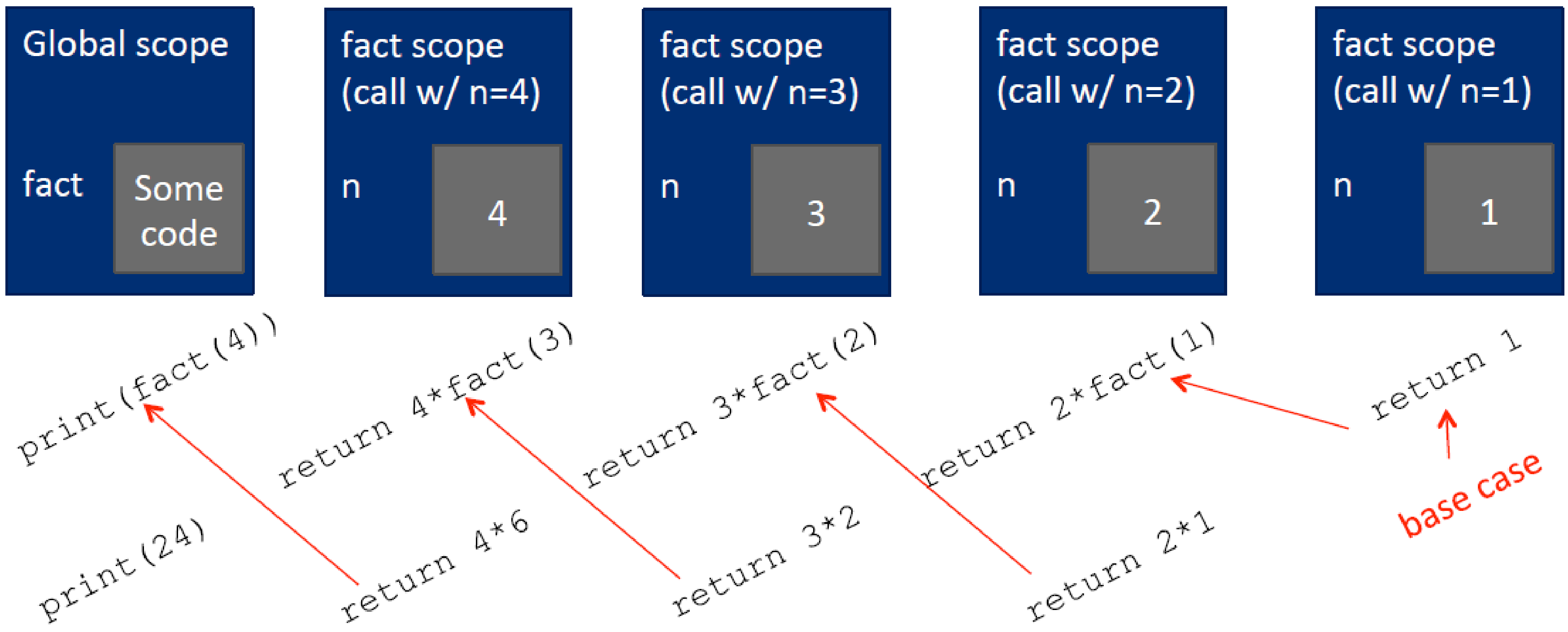
  n = 1 →
  ```
  if n == 1:
      return 1
  ```
  base case

- 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))
```



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 are 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

- $0 + 1 + 2 + 3 + \ldots + n = (n(n+1))/2$
- Proof:
  - If $n = 0$, then LHS is 0 and RHS is $0*1/2 = 0$, so true
  - Assume true for some k, then need to show that

    $$0 + 1 + 2 + \ldots + k + (k+1) = ((k+1)(k+2))/2$$

    - 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

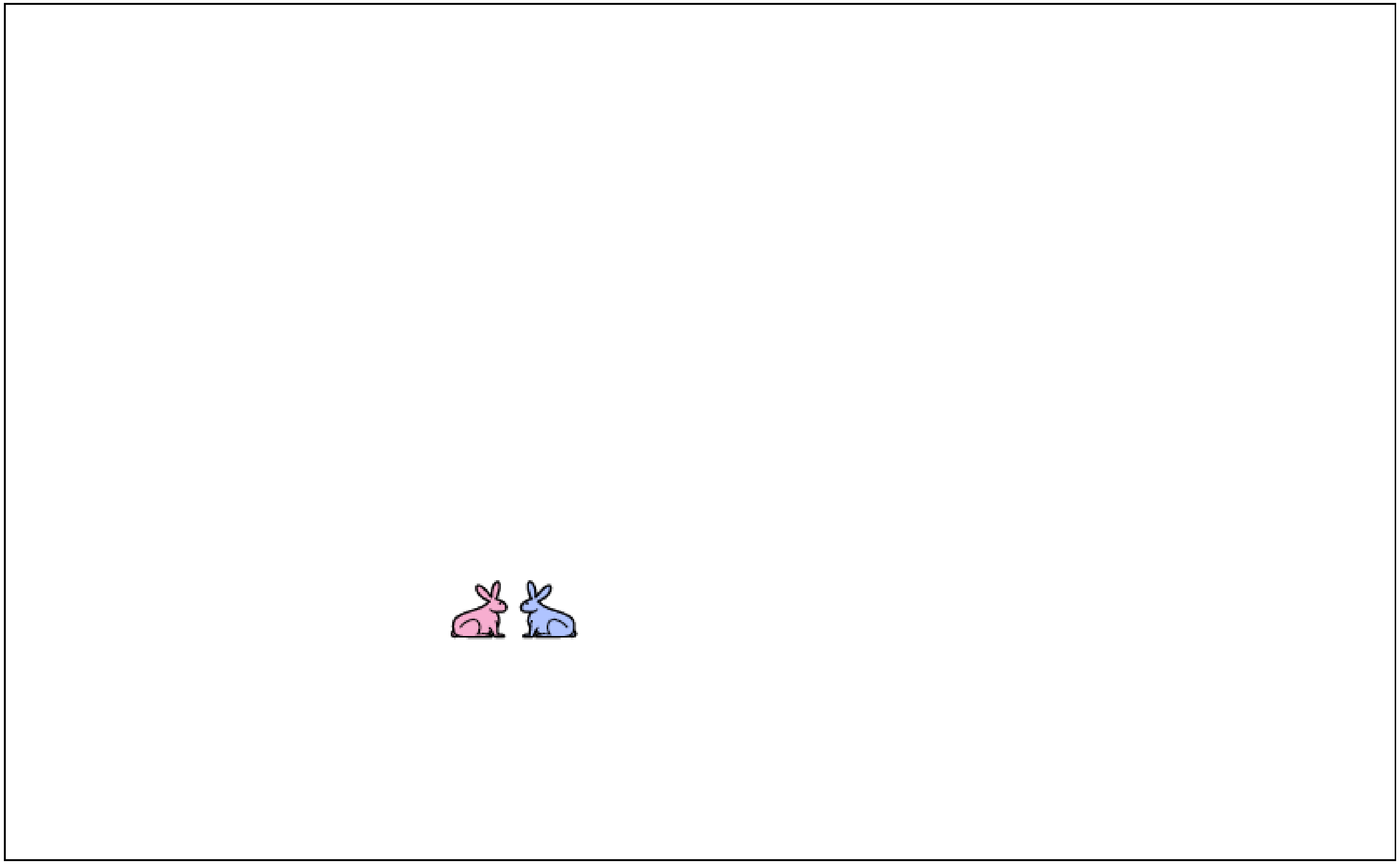
- Having seen a set of examples of different sized stacks, how would you write a program to print out the right set of moves?
- **Think recursively!**
  - Solve a smaller problem
  - Solve a basic problem
  - Solve a smaller problem

TOWERS OF HANOI

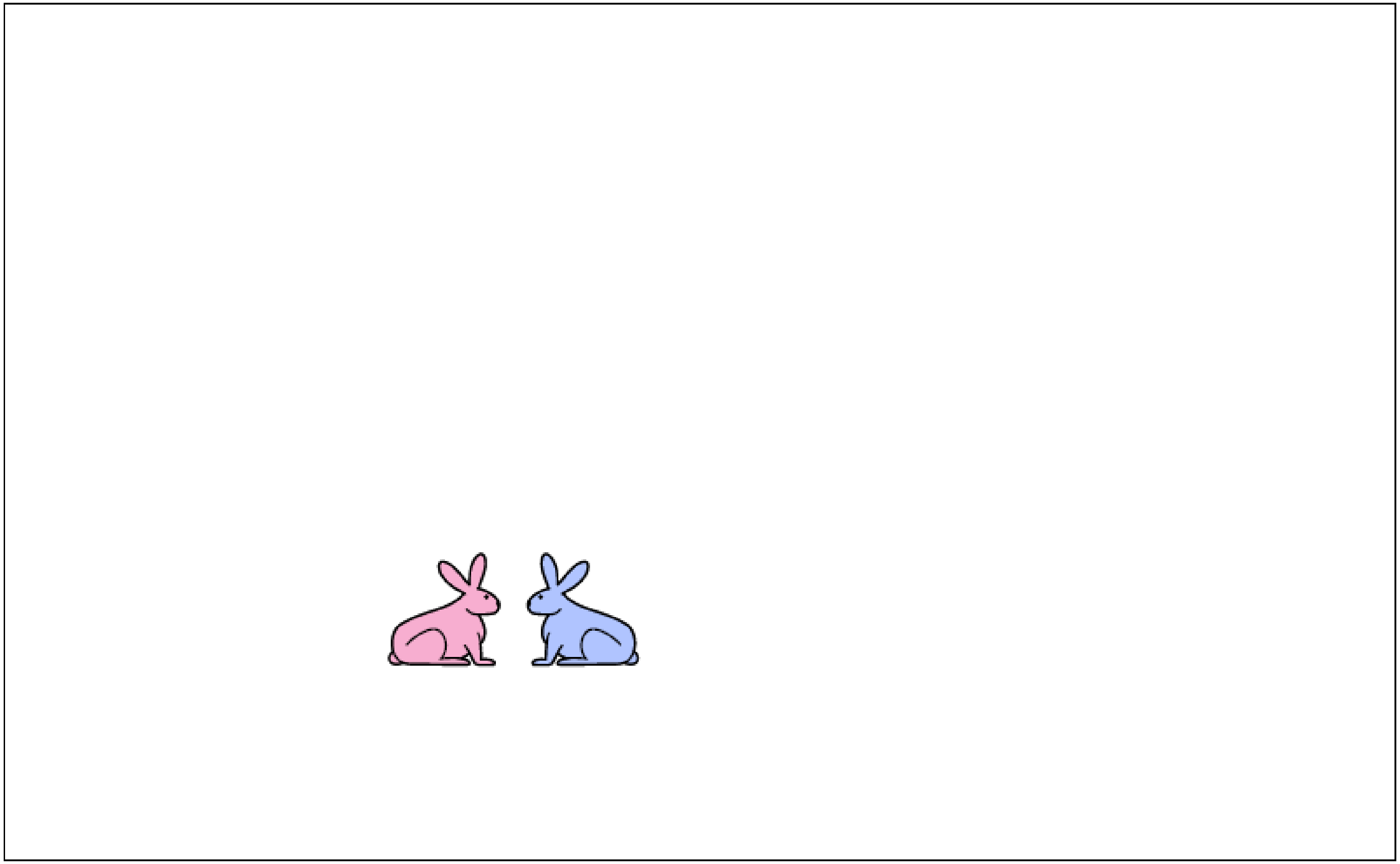
```
def printMove(fr, to):
    print('move from ' + str(fr) + ' to ' + str(to))

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)

print(Towers(4, 'P1', 'P2', 'P3'))
```

RECURSION WITH MULTIPLE BASE CASES

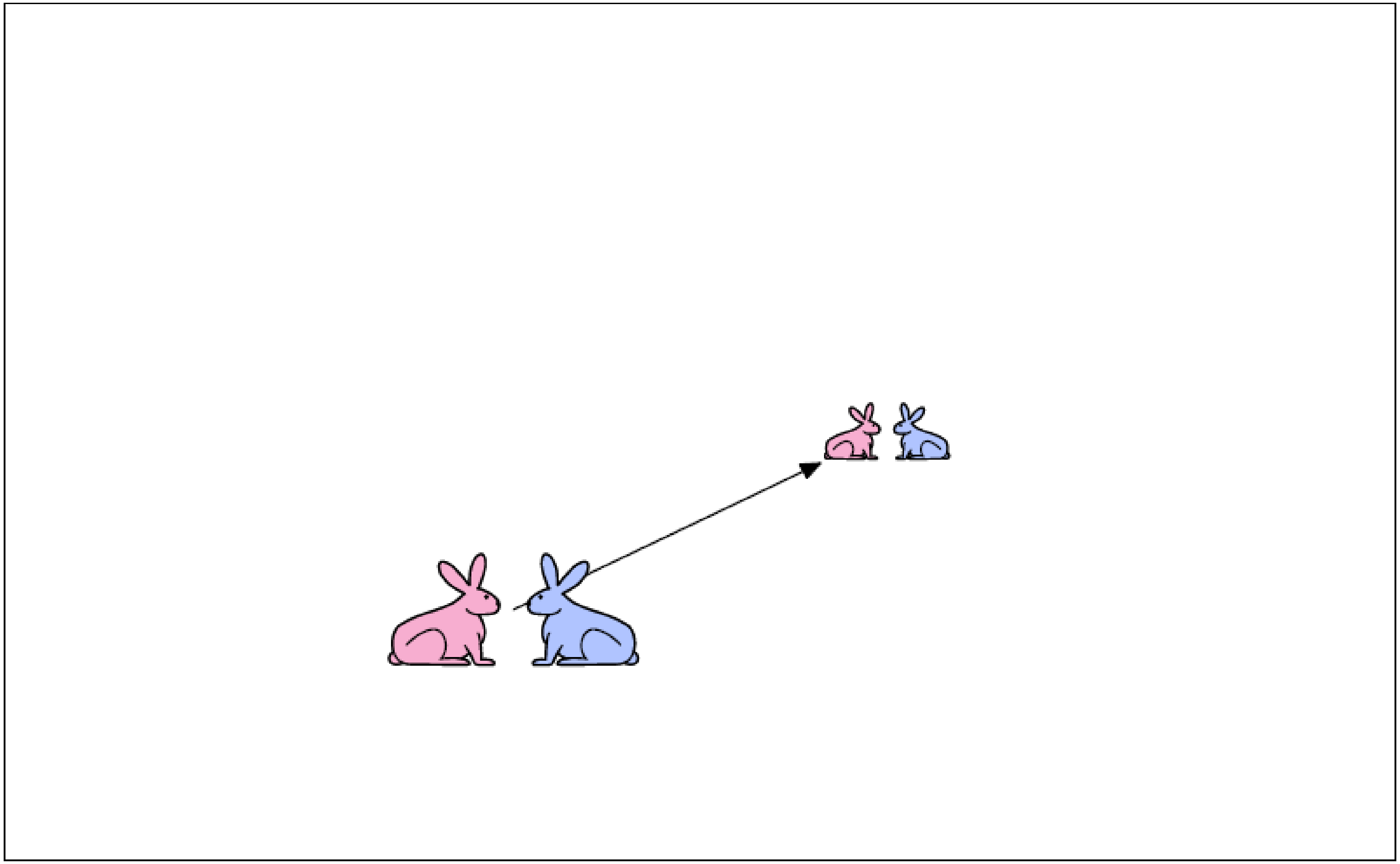
- Fibonacci numbers
  - Leonardo of Pisa (aka Fibonacci) modeled the following challenge
    - Newborn pair of rabbits (one female, one male) are put in a pen
    - Rabbits mate at age of one month
    - Rabbits have a one month gestation period
    - Assume rabbits never die, that female always produces 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?



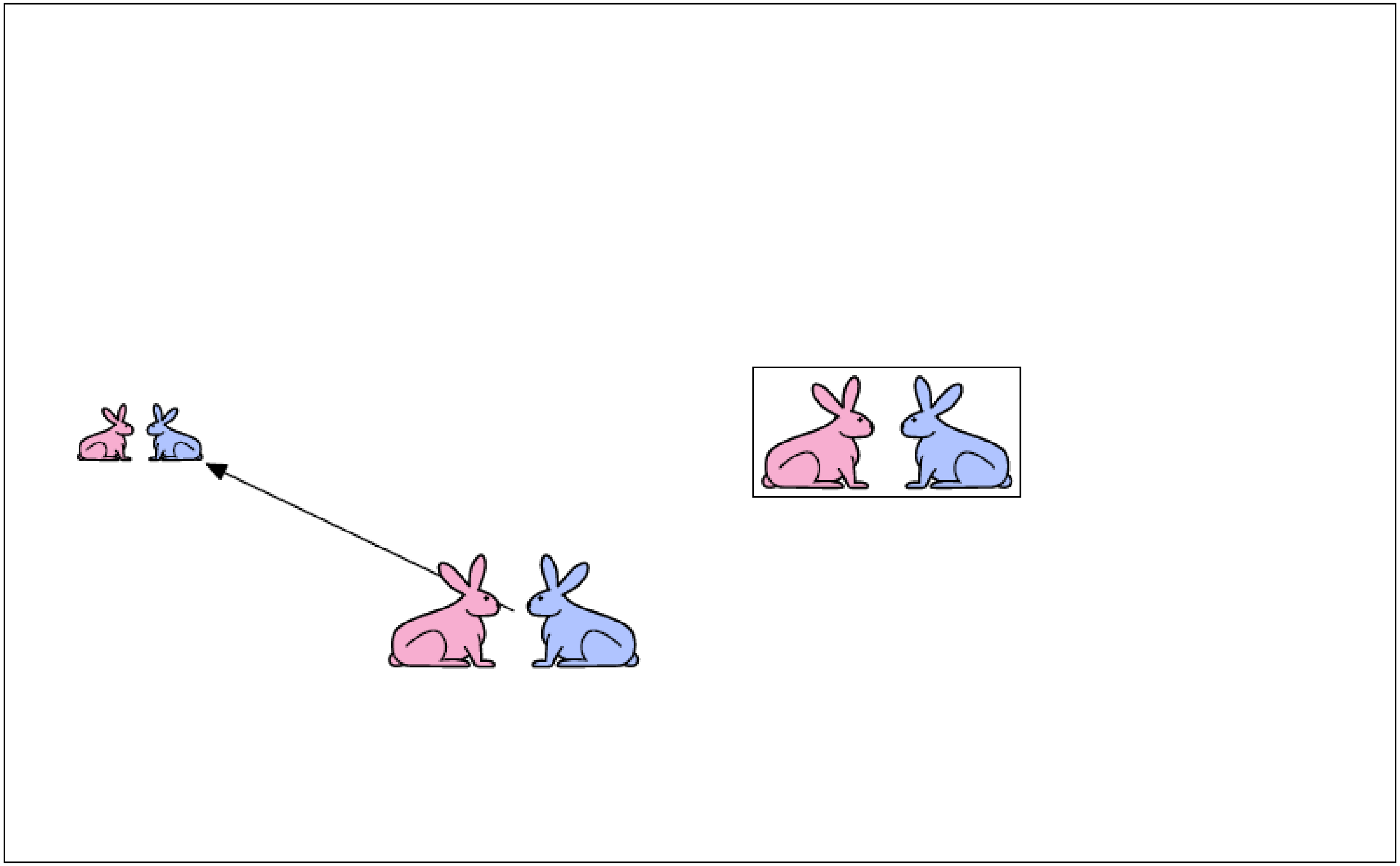
Demo courtesy of Prof. Denny Freeman and Adam Hartz



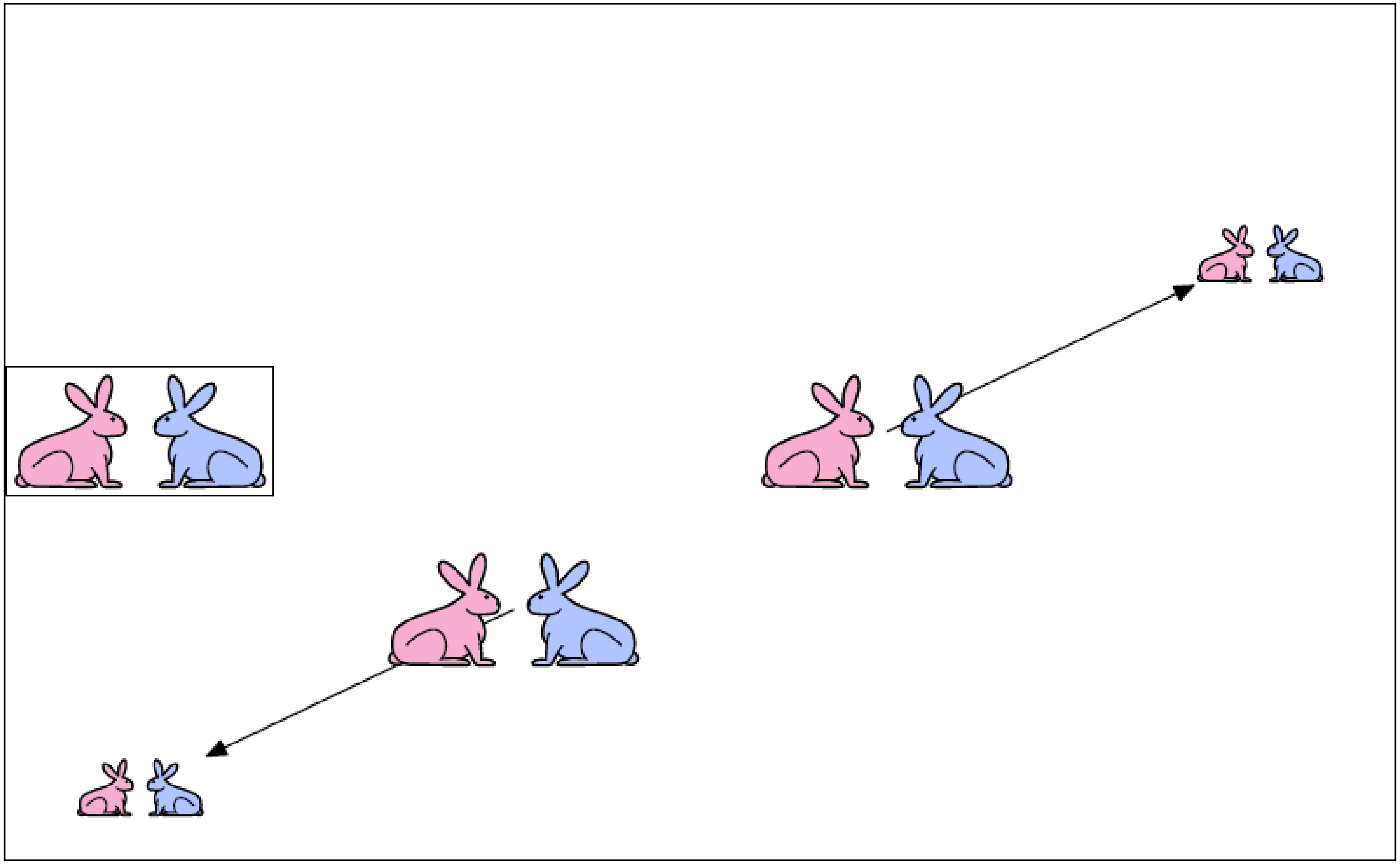
Demo courtesy of Prof. Denny Freeman and Adam Hartz



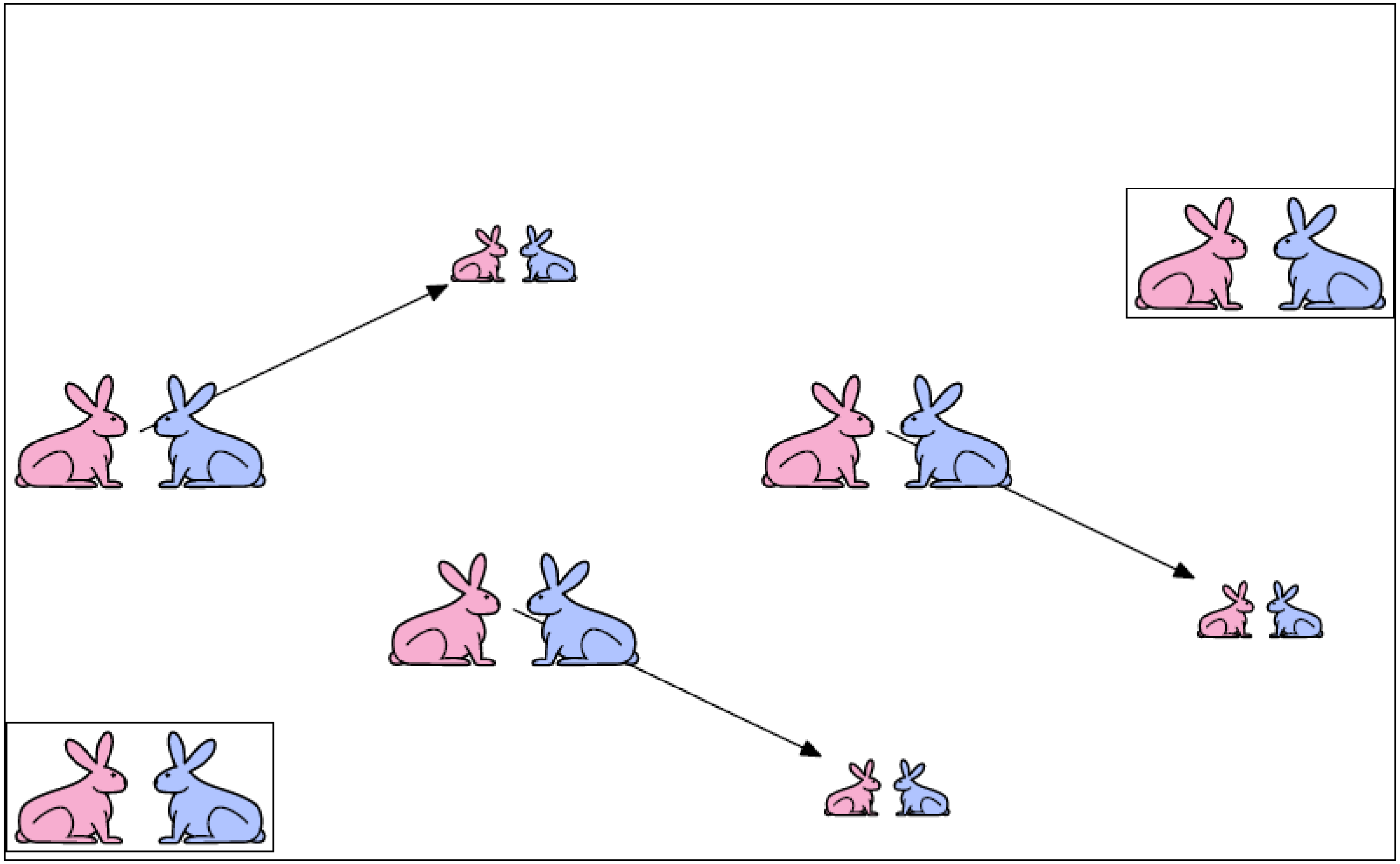
Demo courtesy of Prof. Denny Freeman and Adam Hartz



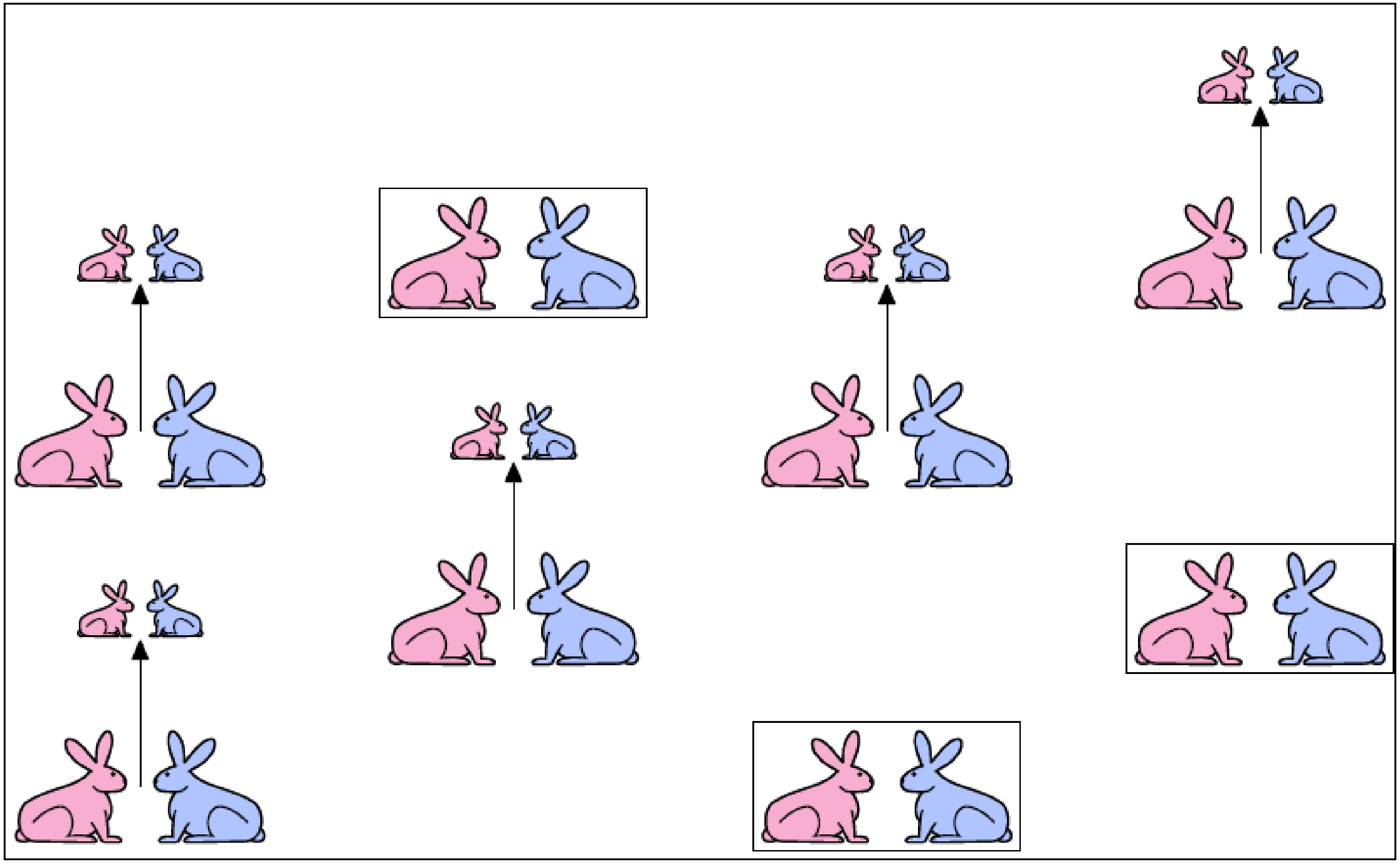
Demo courtesy of Prof. Denny Freeman and Adam Hartz



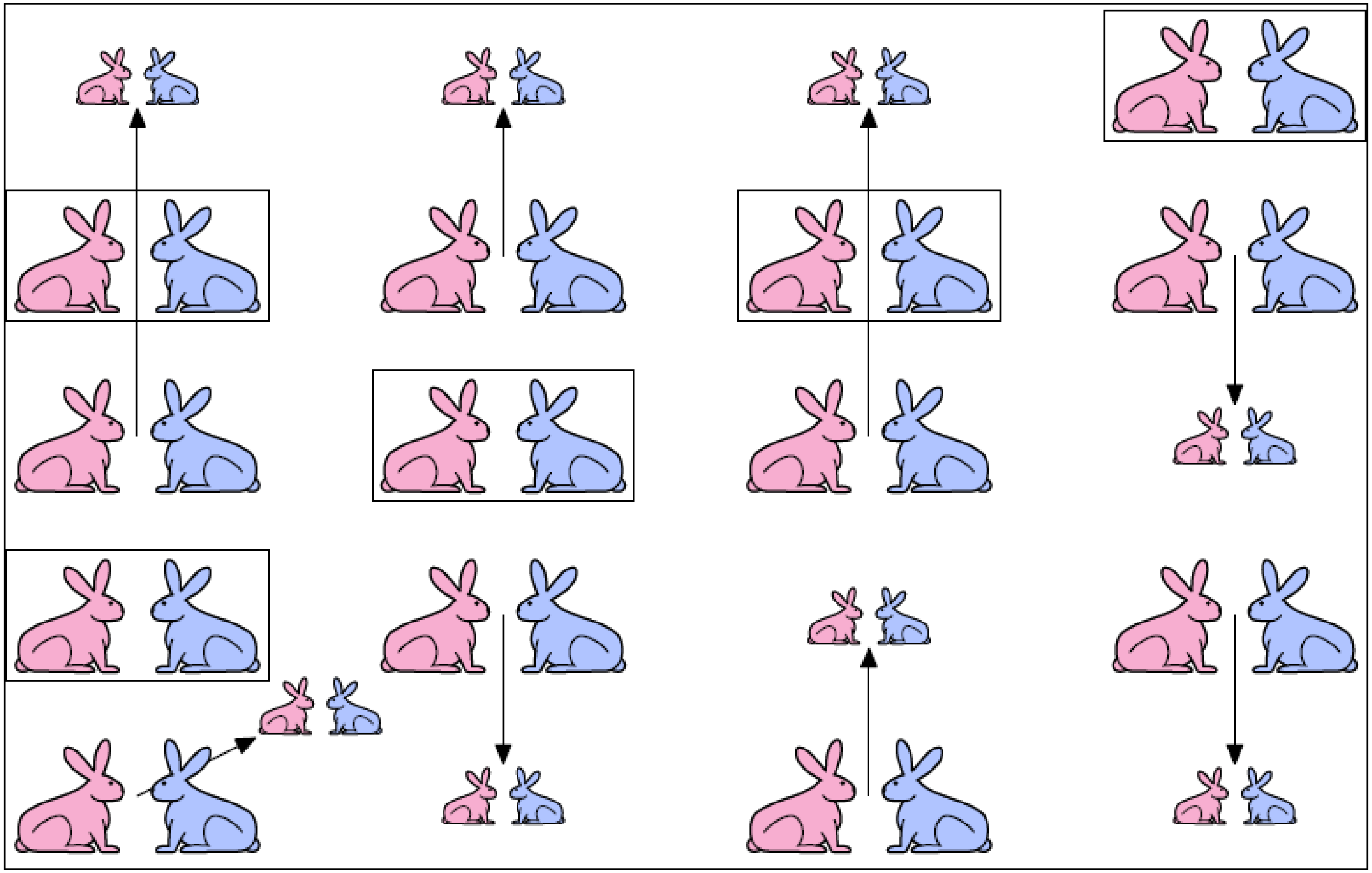
Demo courtesy of Prof. Denny Freeman and Adam Hartz

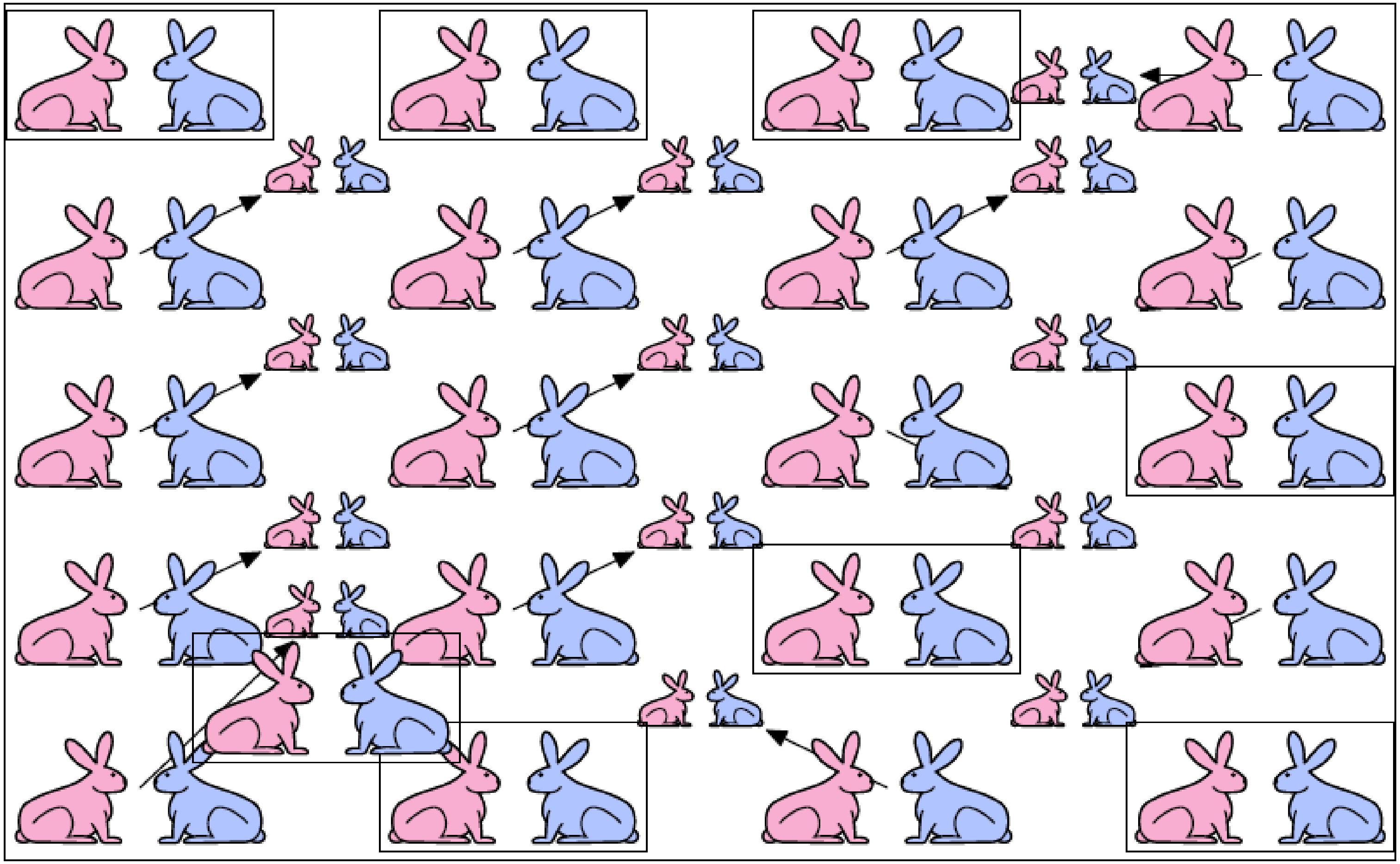


Demo courtesy of Prof. Denny Freeman and Adam Hartz

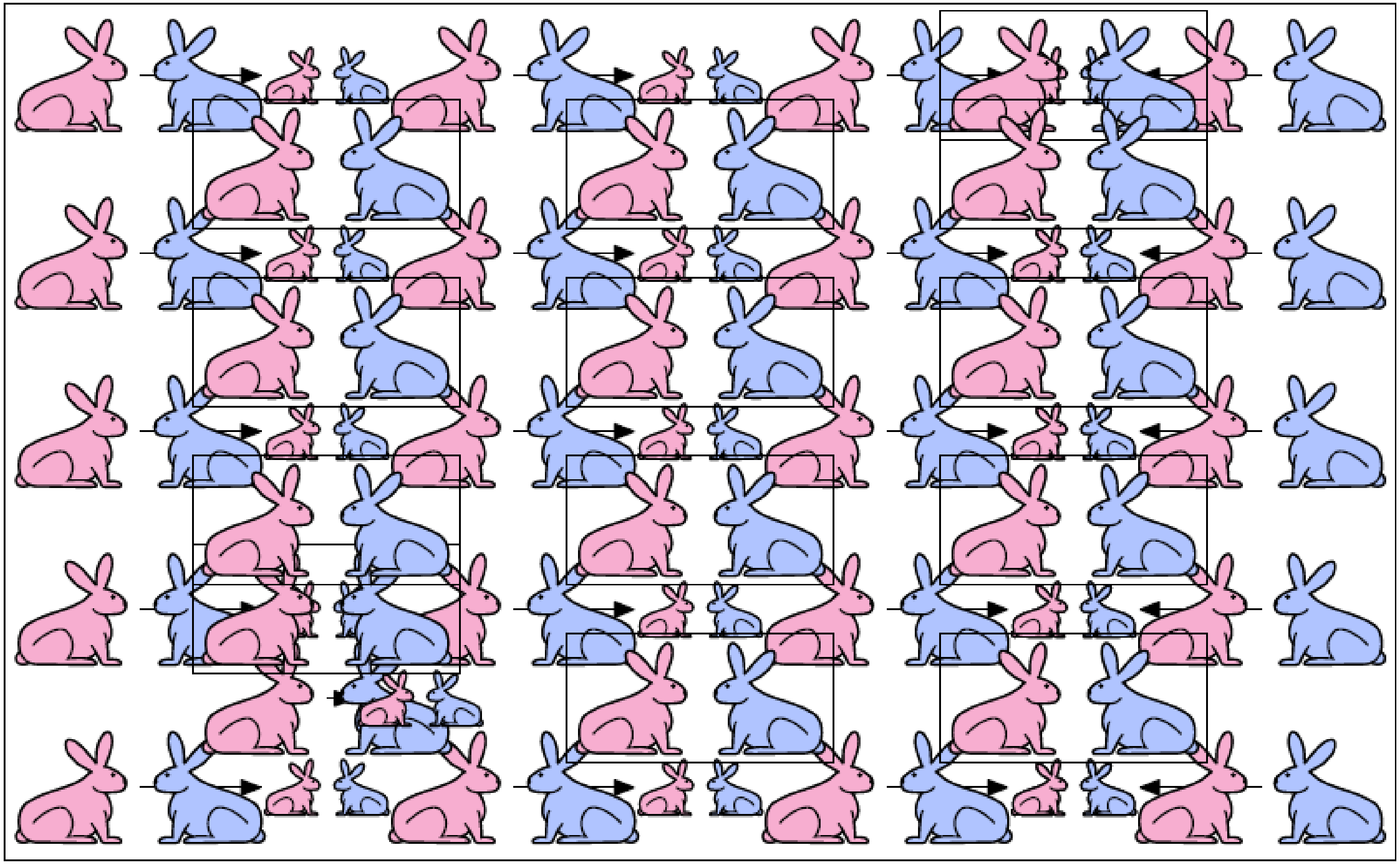


Demo courtesy of Prof. Denny Freeman and Adam Hartz





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

FIBONACCI

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

FIBONACCI

```
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

Image courtesy of wikipedia, in the public domain.

By Larth_Rasnal (Own work) [GFDL (https://www.gnu.org/licenses/fdl-1.3.en.html) or CC BY 3.0 (https://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')`

```
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))

#print(isPalindrome('eve'))
#print(isPalindrome('Able was I, ere I saw Elba'))
#print(isPalindrome('Is this a palindrome'))
```

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

DICTIONARIES

HOW TO STORE STUDENT INFO

- so far, can store using separate lists for every info

```
names = ['Ana', 'John', 'Denise', 'Katy']
grade = ['B', 'A+', 'A', 'A']
course = [2.00, 6.0001, 20.002, 9.01]
```

- a **separate list** for each item
- each list must have the **same length**
- info stored across lists at **same index**, each index refers to info for a different person

HOW TO UPDATE/RETRIEVE STUDENT INFO

```
def get_grade(student, name_list, grade_list, course_list):
    i = name_list.index(student)
    grade = grade_list[i]
    course = course_list[i]
    return (course, grade)
```

- **messy** if have a lot of different info to keep track of
- must maintain **many lists** and pass them as arguments
- must **always index** using integers
- must remember to change multiple lists

A BETTER AND CLEANER WAY – A DICTIONARY

- nice to **index item of interest directly** (not always int)
- nice to use **one data structure**, no separate lists

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
…	…

index element

A dictionary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
…	…

custom index by label element

A PYTHON DICTIONARY

- store pairs of data
  - key
  - value

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

custom index by label

element

empty dictionary

```
my_dict = {}

grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

key1 val1 key2 val2 key3 val3 key4 val4

DICTIONARY LOOKUP

- similar to indexing into a list
- **looks up** the **key**
- **returns** the **value** associated with the key
- if key isn't found, get an error

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

`grades['John']` → evaluates to `'A+'`

`grades['Sylvan']` → gives a `KeyError`

DICTIONARY OPERATIONS

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'
'Sylvan'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- **add** an entry

```
grades['Sylvan'] = 'A'
```

- **test** if key in dictionary

```
'John' in grades        → returns True
'Daniel' in grades      → returns False
```

- **delete** entry

```
del(grades['Ana'])
```

DICTIONARY OPERATIONS

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- get an **iterable that acts like a tuple of all keys** no guaranteed order

  `grades.keys()` → returns `['Denise','Katy','John','Ana']`

- get an **iterable that acts like a tuple of all values**

  `grades.values()` → returns `['A', 'A', 'A+', 'B']`

no guaranteed order

DICTIONARY KEYS and VALUES

- values
  - any type (**immutable and mutable**)
  - can be **duplicates**
  - dictionary values can be lists, even other dictionaries!
- keys
  - must be **unique**
  - **immutable** type (`int`, `float`, `string`, `tuple`, `bool`)
    - actually need an object that is **hashable,** but think of as immutable as all immutable types are hashable
  - careful with `float` type as a key
- **no order** to keys or values!

```
d = {4:{1:0}, (1,3):"twelve", 'const':[3.14,2.7,8.44]}
```

`list` vs `dict`

- **ordered** sequence of elements
- look up elements by an integer index
- indices have an **order**
- index is an **integer**

- **matches** “keys” to “values”
- look up one item by another item
- **no order** is guaranteed
- key can be any **immutable** type

EXAMPLE: 3 FUNCTIONS TO ANALYZE SONG LYRICS

1) create a **frequency dictionary** mapping `str:int`

2) find **word that occurs the most** and how many times
- use a list, in case there is more than one word
- return a tuple `(list,int)` for (words_list, highest_freq)

3) find the **words that occur at least X times**
- let user choose "at least X times", so allow as parameter
- return a list of tuples, each tuple is a `(list, int)` containing the list of words ordered by their frequency
- IDEA: From song dictionary, find most frequent word. Delete most common word. Repeat. It works because you are mutating the song dictionary.

CREATING A DICTIONARY

```
def lyrics_to_frequencies(lyrics):
    myDict = {}
    for word in lyrics:
        if word in myDict:
            myDict[word] += 1
        else:
            myDict[word] = 1
    return myDict
```

can iterate over list

can iterate over keys in dictionary

update value associated with key

```
def lyrics_to_frequencies(lyrics):
    myDict = {}
    for word in lyrics:
        if word in myDict:
            myDict[word] += 1
        else:
            myDict[word] = 1
    return myDict

she_loves_you = ['she', 'loves', 'you', 'yeah', 'yeah',
'yeah','she', 'loves', 'you', 'yeah', 'yeah', 'yeah',
'she', 'loves', 'you', 'yeah', 'yeah', 'yeah',

'you', 'think', "you've", 'lost', 'your', 'love',
'well', 'i', 'saw', 'her', 'yesterday-yi-yay',
"it's", 'you', "she's", 'thinking', 'of',
'and', 'she', 'told', 'me', 'what', 'to', 'say-yi-yay']

beatles = lyrics_to_frequencies(she_loves_you)
print(beatles)
```

USING THE DICTIONARY

```
def most_common_words(freqs):
    values = freqs.values()
    best = max(values)
    words = []
    for k in freqs:
        if freqs[k] == best:
            words.append(k)
    return (words, best)
```

this is an iterable, so can apply built-in function

can iterate over keys in dictionary

LEVERAGING DICTIONARY PROPERTIES

```
def words_often(freqs, minTimes):
    result = []
    done = False
    while not done:
        temp = most_common_words(freqs)
        if temp[1] >= minTimes:
            result.append(temp)
            for w in temp[0]:
                del(freqs[w])
        else:
            done = True
    return result


print(words_often(beatles, 5))
```

can directly mutate dictionary; makes it easier to iterate

```python
def most_common_words(freqs):
    values = freqs.values()
    best = max(freqs.values())
    words = []
    for k in freqs:
        if freqs[k] == best:
            words.append(k)
    return (words, best)
```

```python
def words_often(freqs, minTimes):
    result = []
    done = False
    while not done:
        temp = most_common_words(freqs)
        if temp[1] >= minTimes:
            result.append(temp)
            for w in temp[0]:
                del(freqs[w])  #remove word from dict
        else:
            done = True
    return result

#print(words_often(beatles, 5))
```

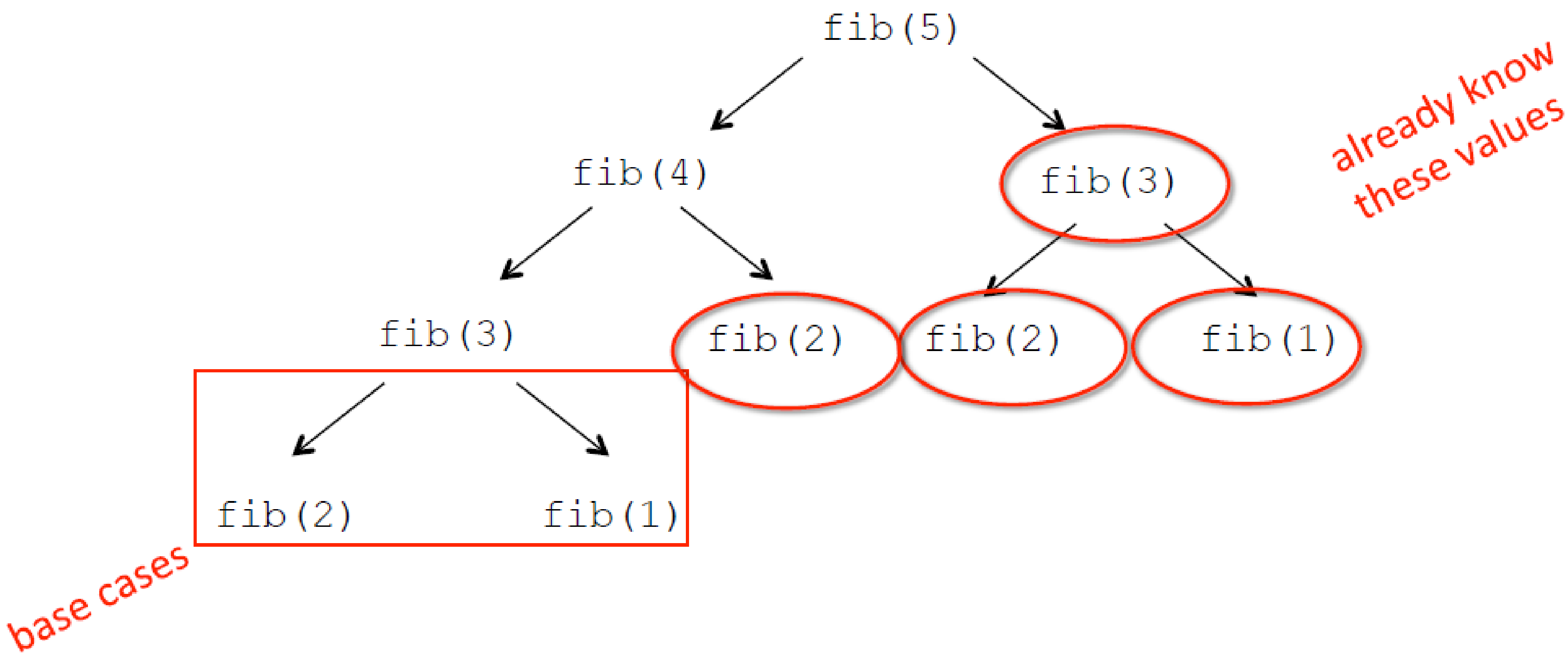
FIBONACCI RECURSIVE CODE

```
def fib(n):
    if n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return fib(n-1) + fib(n-2)
```

- two base cases
- calls itself twice
- this code is inefficient

INEFFICIENT FIBONACCI

`fib(n) = fib(n-1) + fib(n-2)`



- **recalculating** the same values many times!
- could keep **track** of already calculated values

FIBONACCI WITH A DICTIONARY

```
def fib_efficient(n, d):
    if n in d:
        return d[n]
    else:
        ans = fib_efficient(n-1, d) + fib_efficient(n-2, d)
        d[n] = ans
        return ans

d = {1:1, 2:2}
print(fib_efficient(6, d))
```

Method sometimes called " memoization"

Initialize dictionary with base cases

- do a **lookup first** in case already calculated the value
- **modify dictionary** as progress through function calls

```
def fib_efficient(n, d):
    if n in d:
        return d[n]
    else:
        ans = fib_efficient(n-1, d)+fib_efficient(n-2, d)
        d[n] = ans
        return ans

d = {1:1, 2:2}

argToUse = 34
#print("")
#print('using fib')
#print(fib(argToUse))
#print("")
#print('using fib_efficient')
#print(fib_efficient(argToUse, d))
```

EFFICIENCY GAINS

- Calling fib(34) results in 11,405,773 recursive calls to the procedure
- Calling fib_efficient(34) results in 65 recursive calls to the procedure
- Using dictionaries to capture intermediate results can be very efficient
- But note that this only works for procedures without side effects (i.e., the procedure will always produce the same result for a specific argument independent of any other computations between calls)