ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Lecture 3: Functions

Contents

- Functions
- Specifications
- Keywords: `return` vs `print`
- Variable scope
- Default values
- Lambda functions
- Recursion

Opening Problem

- Find the sum of integers from 1 to 10, from 20 to 37, and from 35 to 49, respectively.

Problem

```
sum = 0
for i in range(1, 10):
    sum += i
print("Sum from 1 to 10 is", sum)

sum = 0
for i in range(20, 37):
    sum += i
print("Sum from 20 to 37 is", sum)

sum = 0
for i in range(35, 49):
    sum += i
print("Sum from 35 to 49 is", sum)
```

Problem

```
sum = 0
for i in range(1, 10):
    sum += i
print("Sum from 1 to 10 is", sum)
```

```
sum = 0
for i in range(20, 37):
    sum += i
print("Sum from 20 to 37 is", sum)
```

```
sum = 0
for i in range(35, 49):
    sum += i
print("Sum from 35 to 49 is", sum)
```

Solution

```
def sum(i1, i2):
    result = 0
    for i in range(i1, i2):
        result += i
    return result
```

```
def main():
    print("Sum from 1 to 10 is", sum(1, 10))
    print("Sum from 20 to 37 is", sum(20, 37))
    print("Sum from 35 to 49 is", sum(35, 49))

main() # Call the main function
```

Defining Functions

- A function is a collection of statements that are grouped together to perform an operation. In Python a function is some reusable code that takes arguments(s) as input does some computation and then returns a result or results
- Function characteristics:
  - has a **name**
  - has **parameters** (0 or more)
  - has a **docstring** (optional but recommended)
  - has a **body**
  - **returns** something

Python Functions

- There are two kinds of functions in Python
- Built-in functions that are provided as part of Python - input(), type(), float(), max(), min(), int(), str(), …
- Functions (user defined) that we define ourselves and then use
- We treat the built-in function names like reserved words (i.e. we avoid them as variable names)

Built-in functions

Type conversions

- can **convert object of one type to another**
- When you put an integer and floating point in an expression the integer is implicitly converted to a float
- You can control this with the built in functions int() and float()

```
>>> print (float(99) / 100)
0.99
>>> i = 42
>>> type(i)
<class 'int'>
>>> f = float(i)
>>> print (f)
42.0
>>> type(f)
<class 'float'>
>>> print (1 + 2 * float(3)  – 5)
2.0
>>>
```

Built-in functions

String Conversions

- You can also use int() and float() to convert between strings and integers
- You will get an error if the string does not contain numeric characters

```
>>> sval = '123'
>>> type(sval)
<class 'str'>
>>> print (sval + 1)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    print (sval + 1)
TypeError: can only concatenate str (not "int") to
str
>>> ival = int(sval)
>>> type(ival)
<class 'int'>
>>> print (ival + 1)
124
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    niv = int(nsv)
ValueError: invalid literal for int() with base 10:
'hello bob'
```

User-defined functions

- We create a new function using the def keyword followed by the function name, optional parameters in parenthesis, and then we add a colon.
- We indent the body of the function
- This defines the function but *does not* execute the body of the function

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

How to write a function

keyword

name

parameters or arguments

specification, docstring

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
```

body

```
    print("inside is_even")
    return i%2 == 0
```

later in the code, you call the function using its name and values for parameters

```
is_even(3)
```

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
```

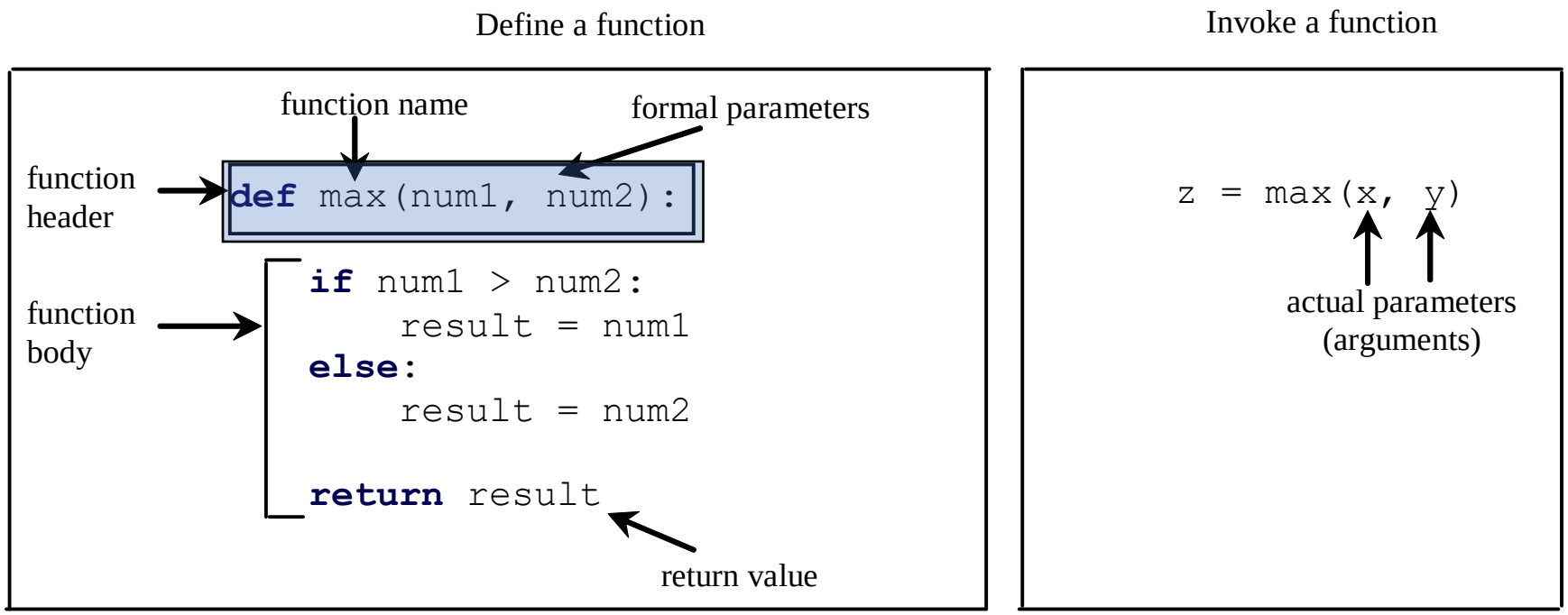
run some commands

keyword

expression to evaluate and return

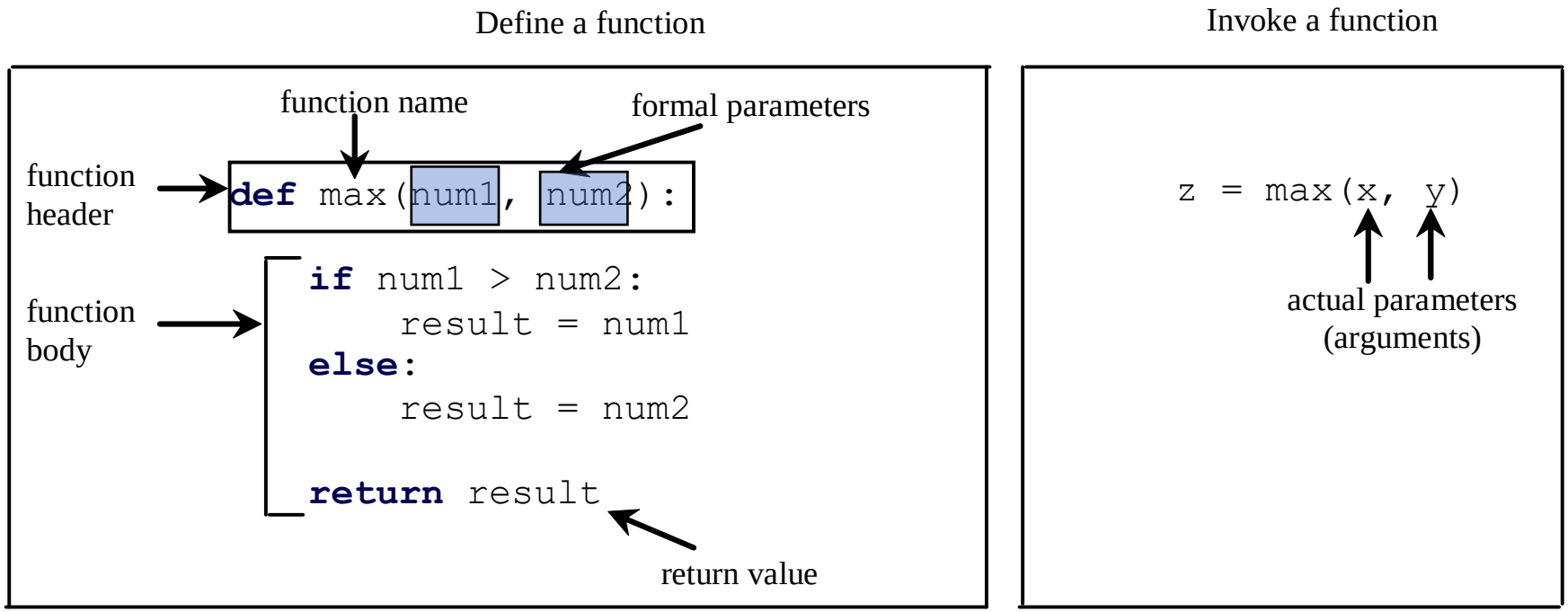
Function Header

A function contains a header and body. The header begins with the **def** keyword, followed by function's name and parameters, followed by a colon.



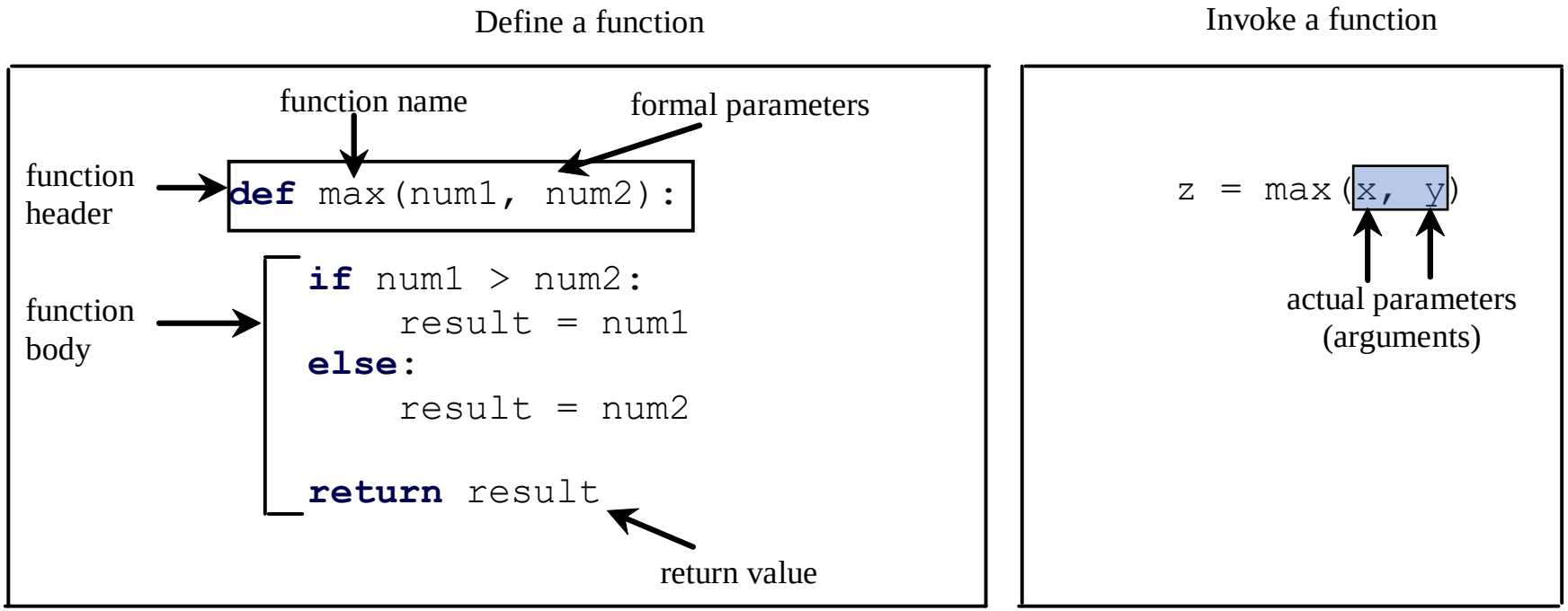
Formal Parameters

- The variables defined in the function header are known as *formal parameters*.
- A *formal parameter* is a variable which we use in the function definition that is a "handle" that allows the code in the function to access the arguments for a particular function invocation.



Actual Parameters

When a function is invoked, you pass a value to the parameter. This value is referred to as *actual parameter or argument*.

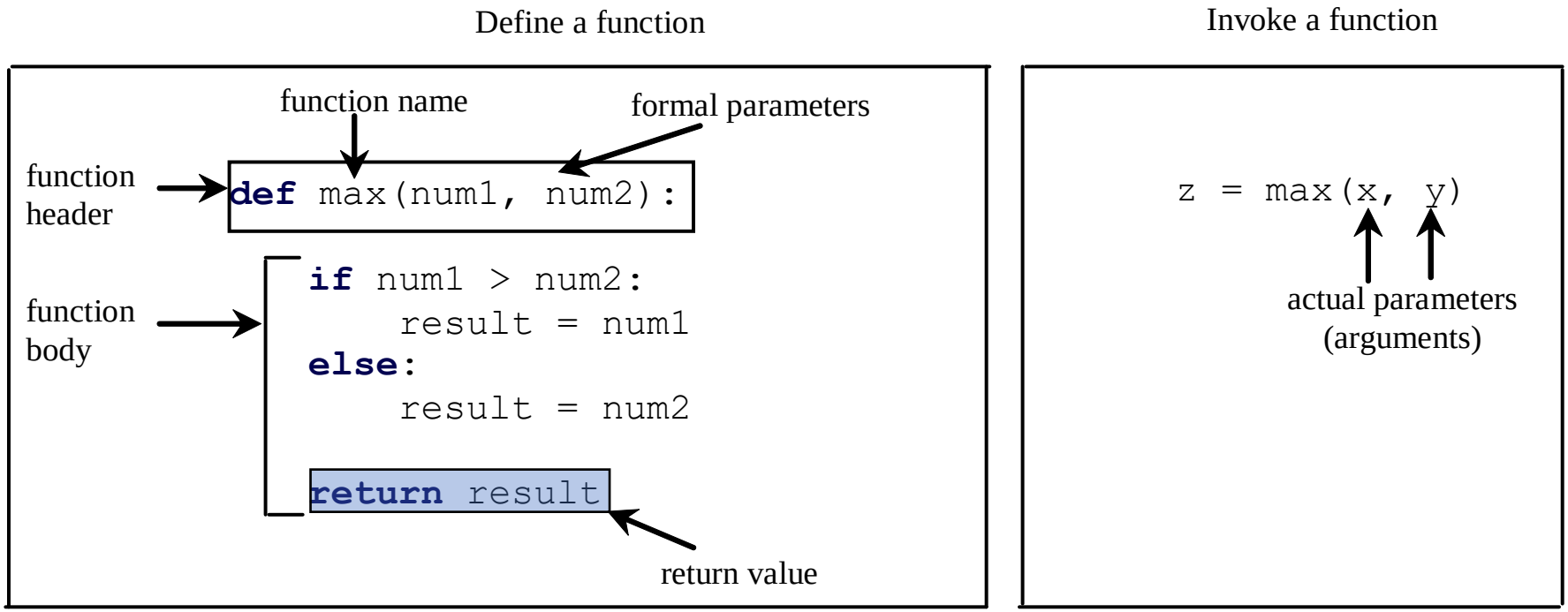


Arguments

- An argument is a value we pass into the function as its input when we call the function
- We use arguments so we can direct the function to do different kinds of work when we call it at different times
- We put the arguments in parenthesis after the name of the function

Return Value

Often a function will take its arguments, do some computation and return a value to be used as the value of the function call in the calling expression. The return keyword is used for this.



Return Value

- A “fruitful” function is one that produces a result (or a return value)
- The return statement ends the function execution and “sends back” the result of the function

```
def greet(lang):
      if lang == 'es':
        return('Hola ')
      elif lang == 'fr':
        return('Bonjour ')
       else:
   return('Hello ‘)

print(greet('en'),'Glenn')
Hello Glenn
print(greet('es'),'Sally')
Hola Sally
print(greet('fr'),'Michael')
Bonjour Michael
```

Void Functions

- When a function does not return a value, we call it a "void" function
- Void functions are "not fruitful"

Multiple Parameters / Arguments

- We can define more than one parameter in the function definition
- We simply add more arguments when we call the function
- We match the number and order of arguments and parameters

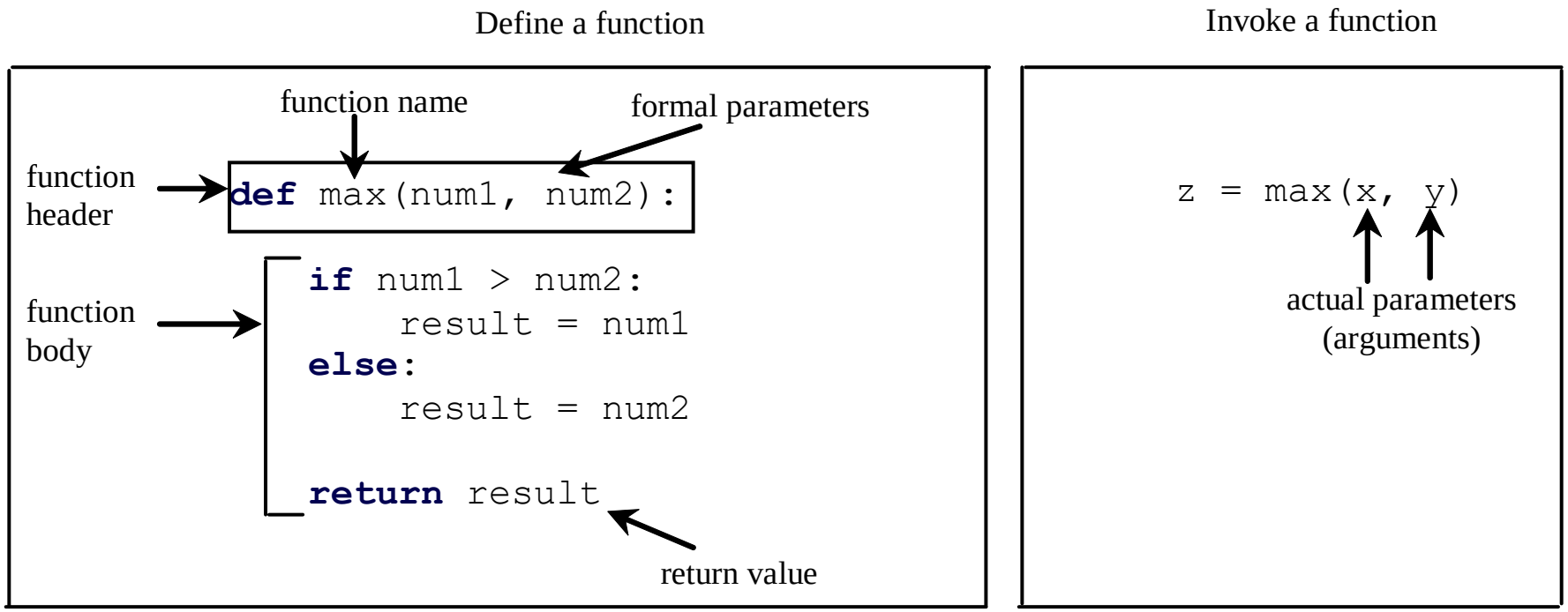
```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)
8
```

Definitions and Uses

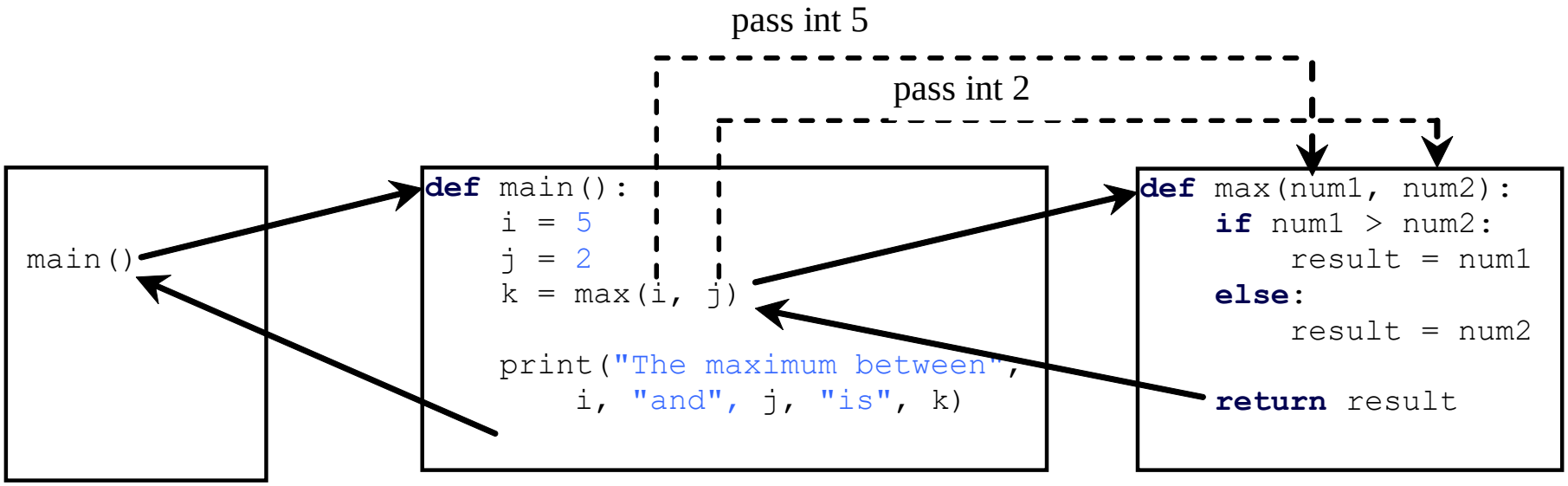
- Once we have defined a function, we can call (or invoke) it as many times as we like
- This is the store and reuse pattern

Calling Functions

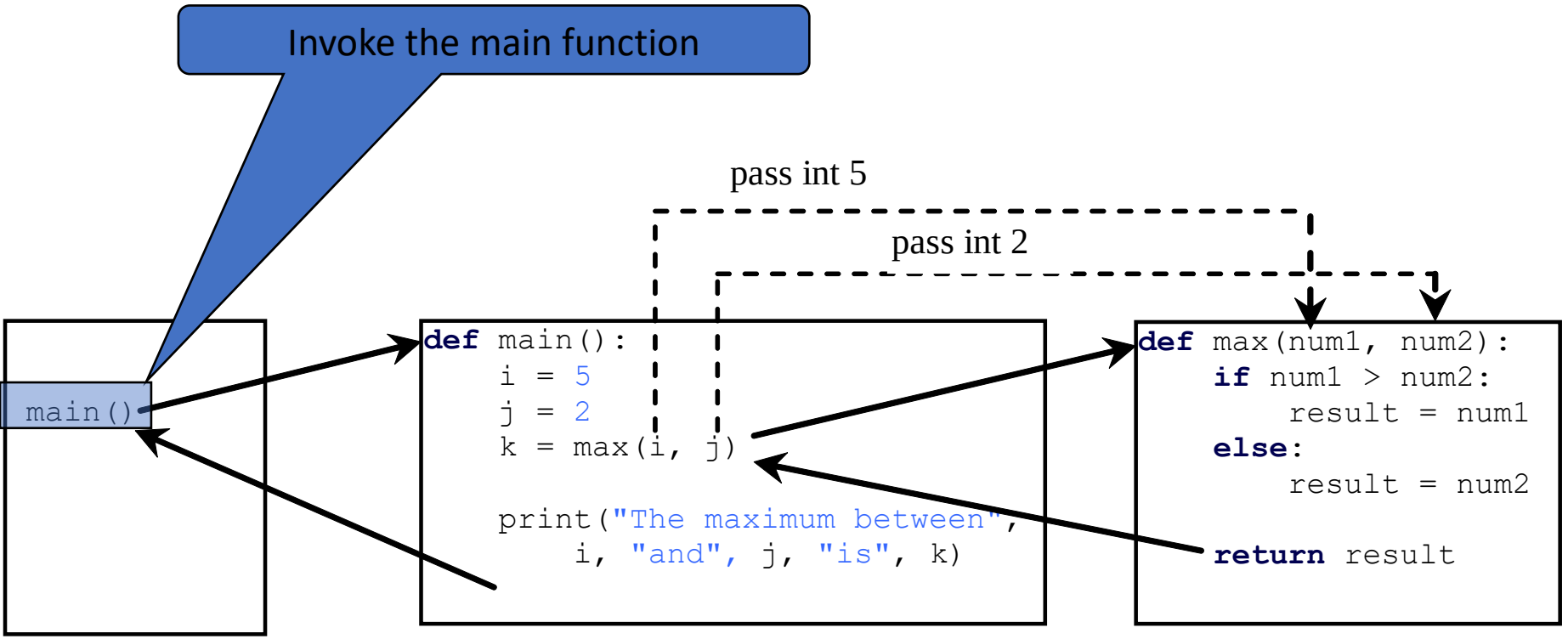
- Testing the max function
- This program demonstrates calling a function max to return the largest of the int values



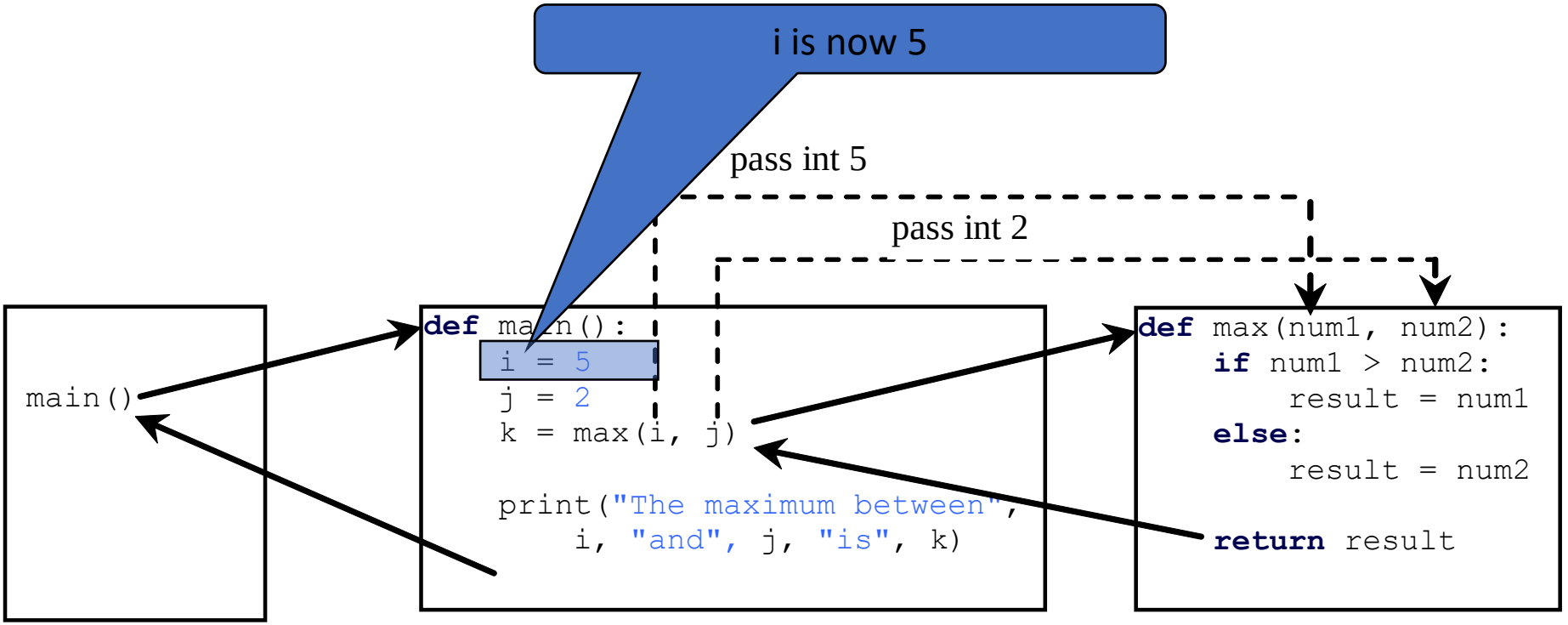
Calling Functions, cont.



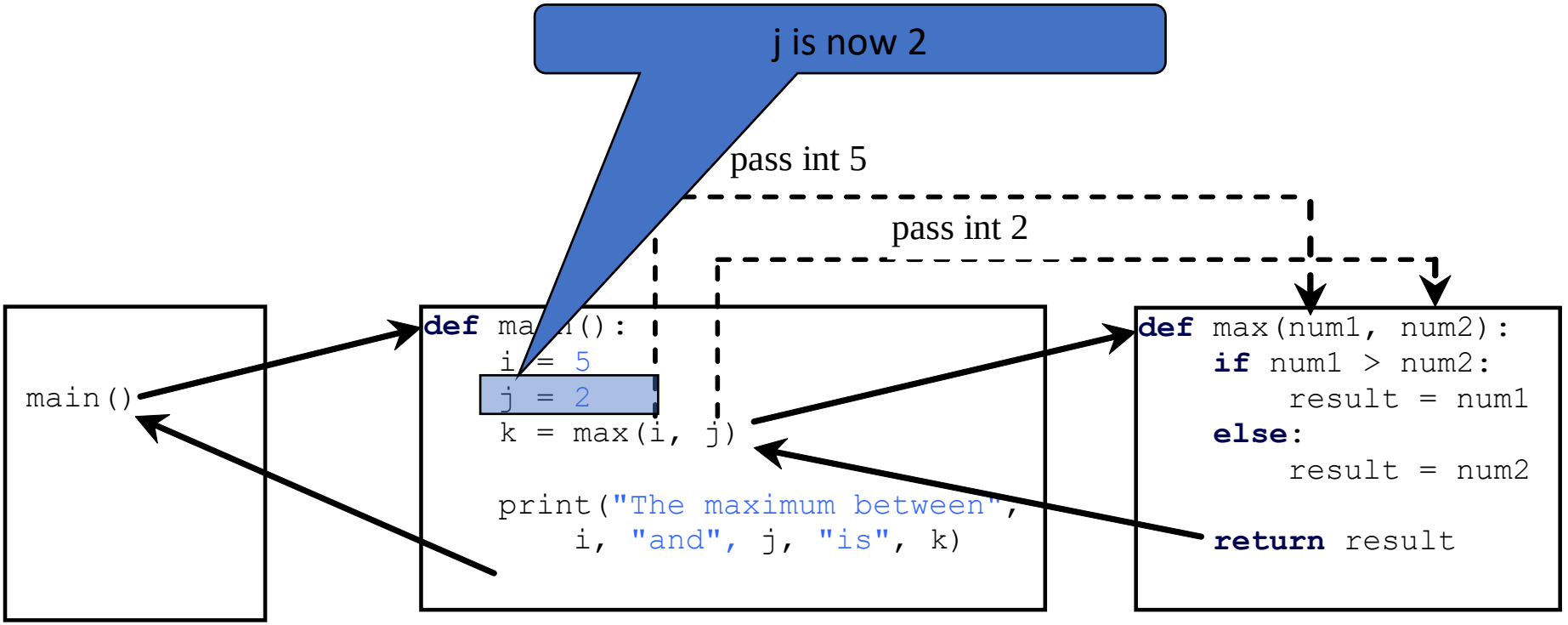
Trace Function Invocation



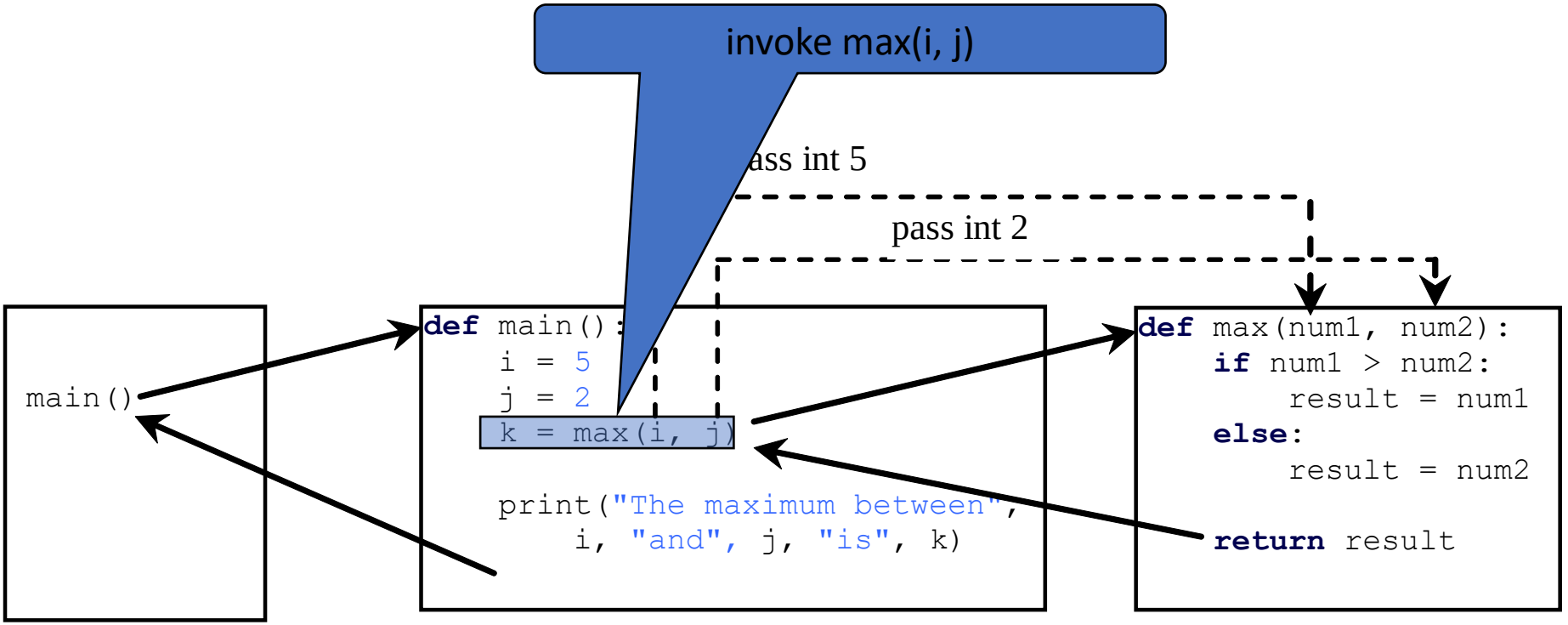
Trace Function Invocation



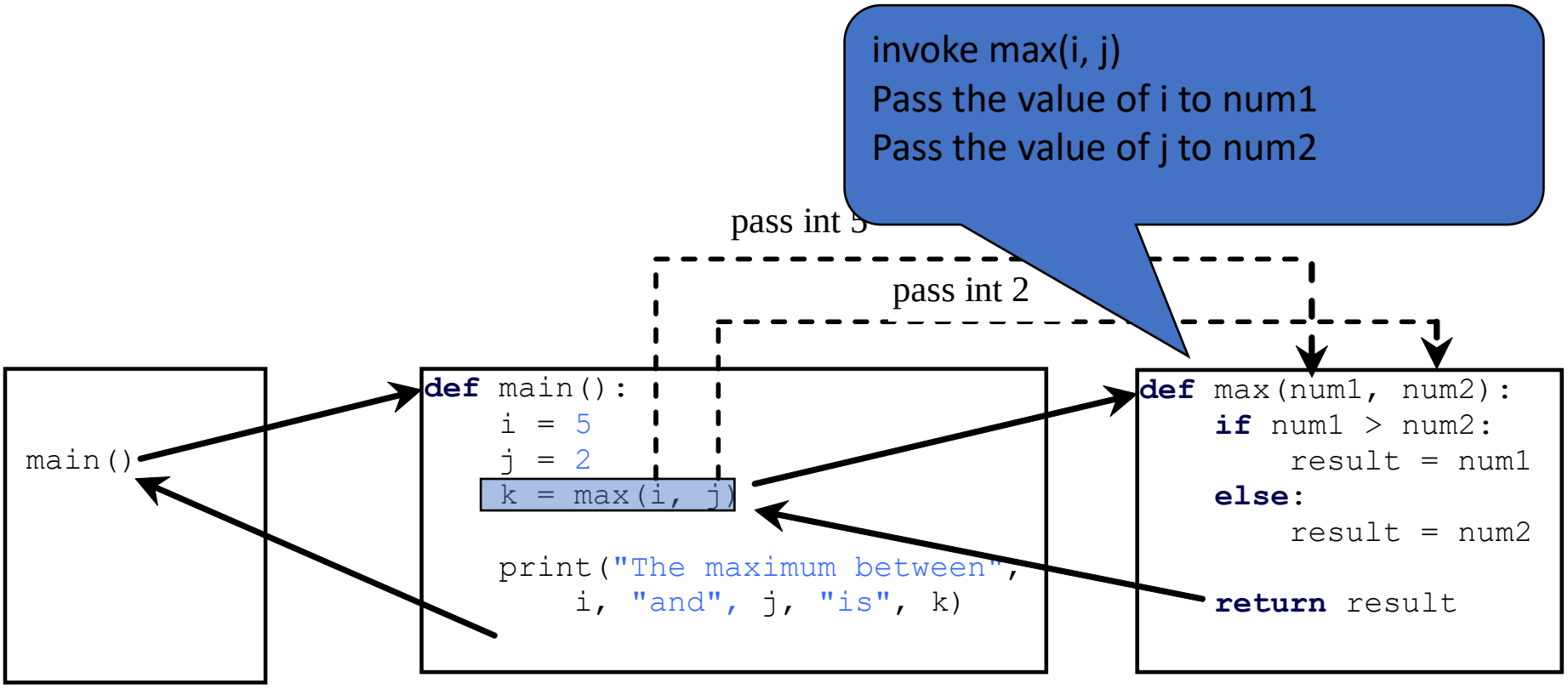
Trace Function Invocation



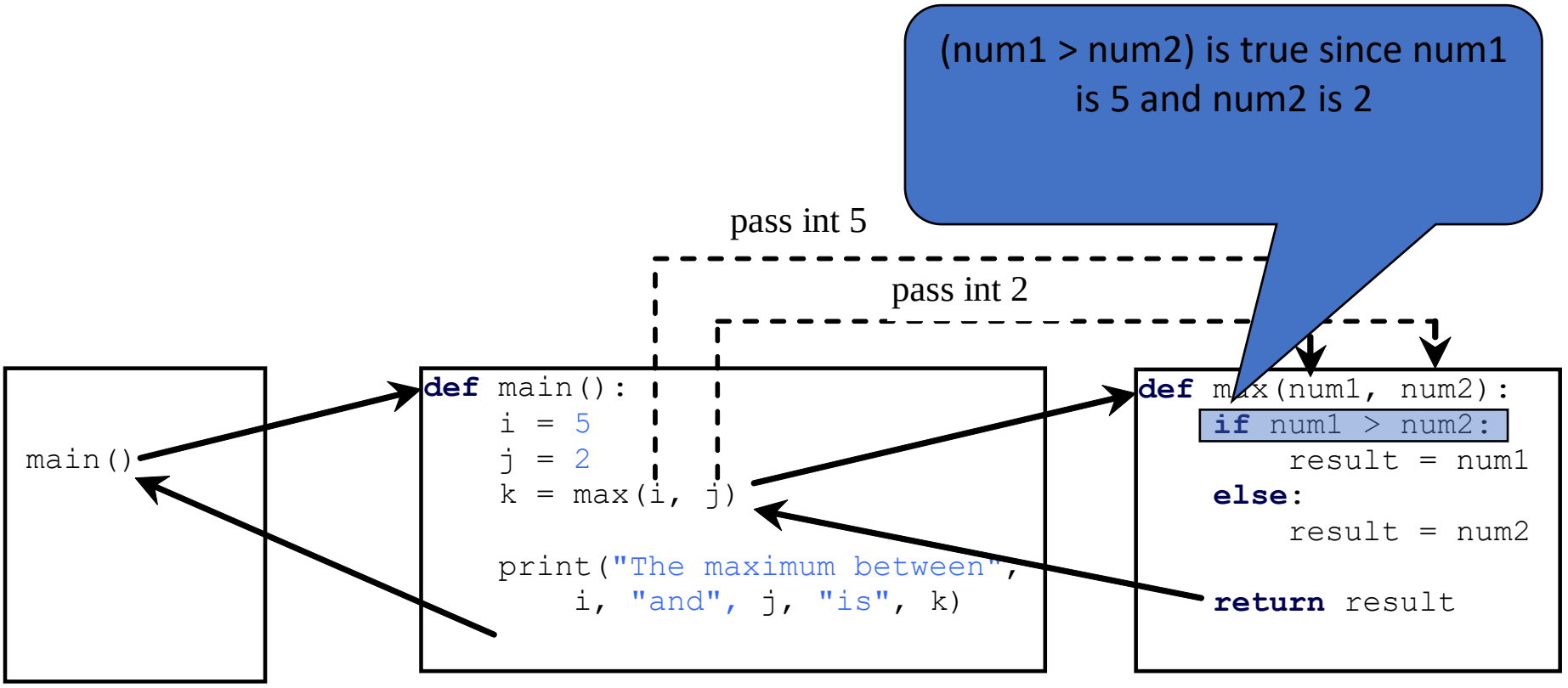
Trace Function Invocation



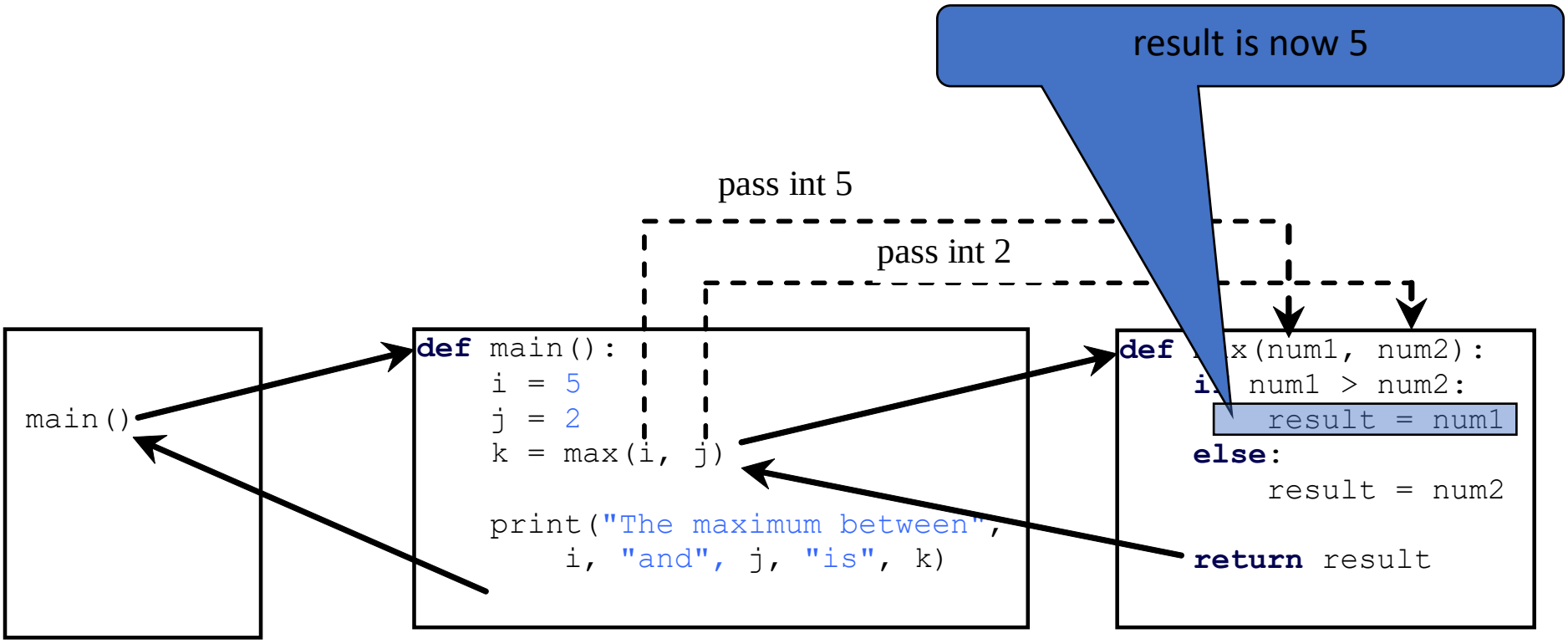
Trace Function Invocation



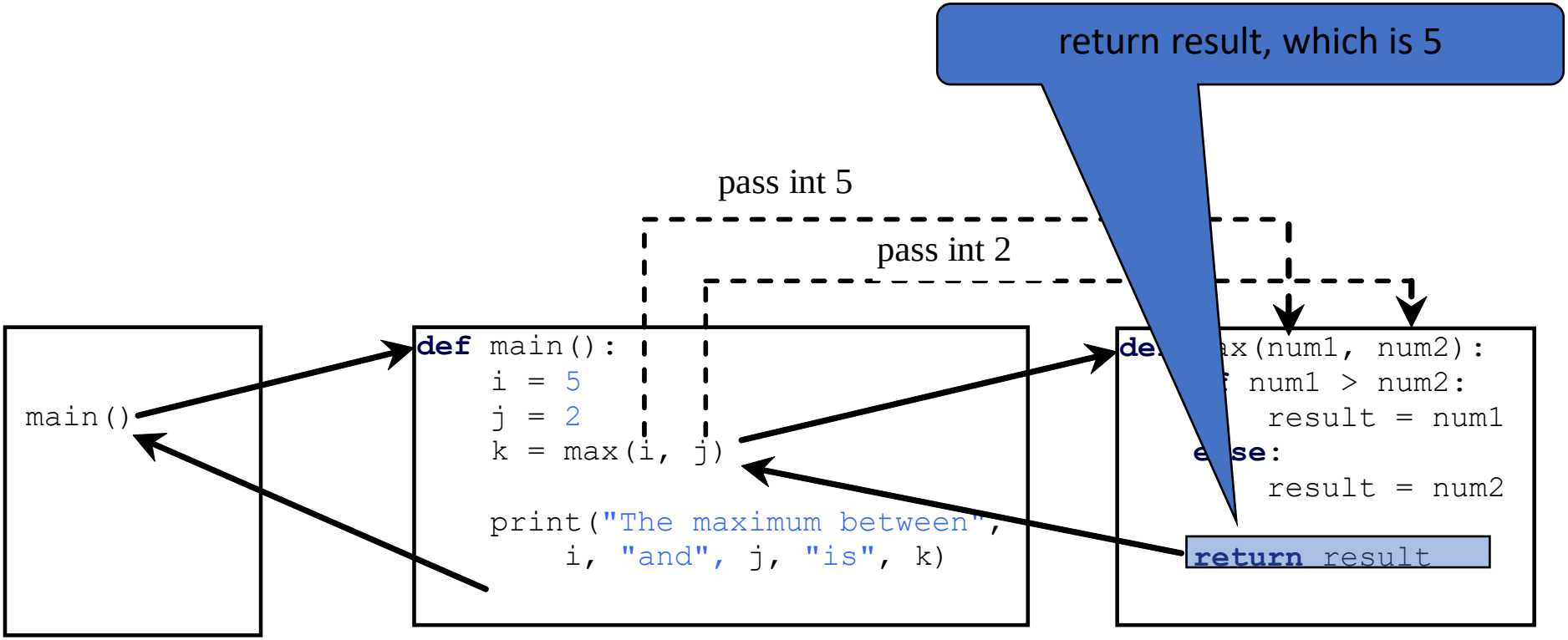
Trace Function Invocation



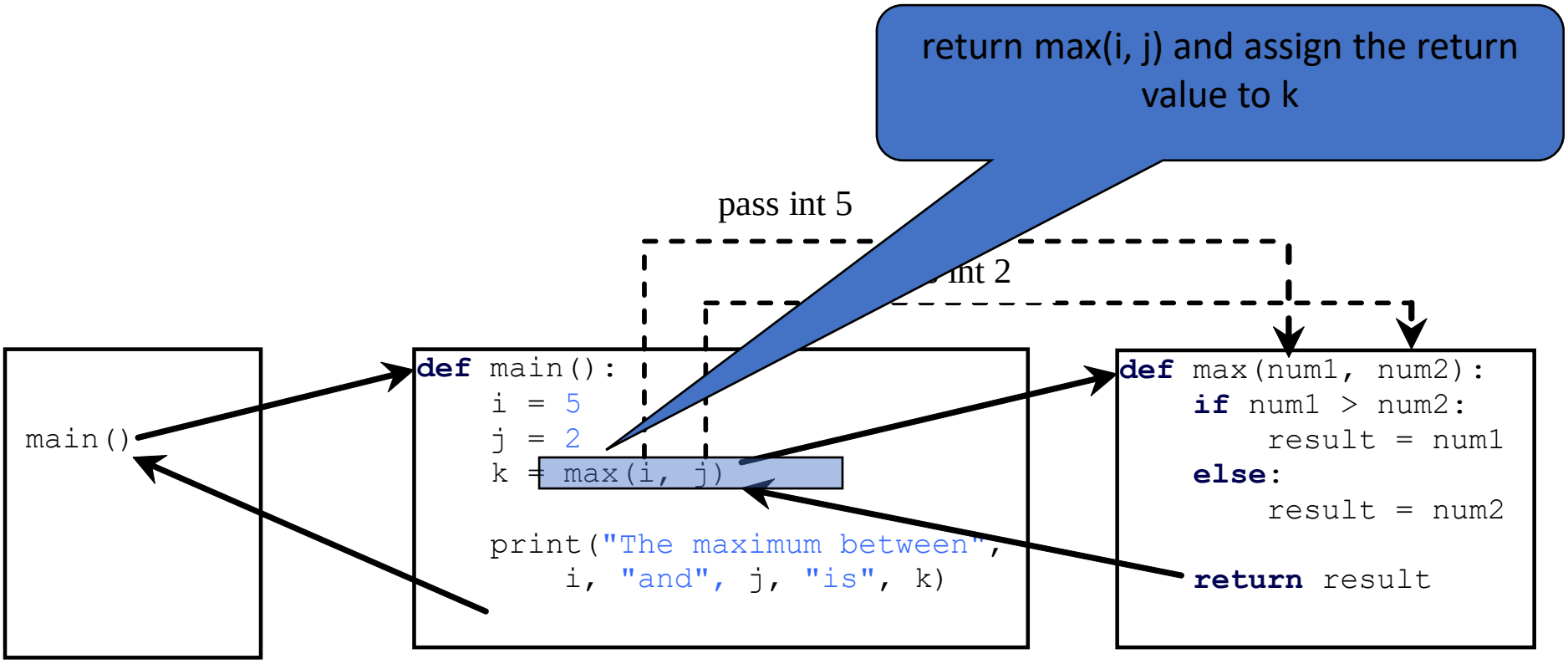
Trace Function Invocation



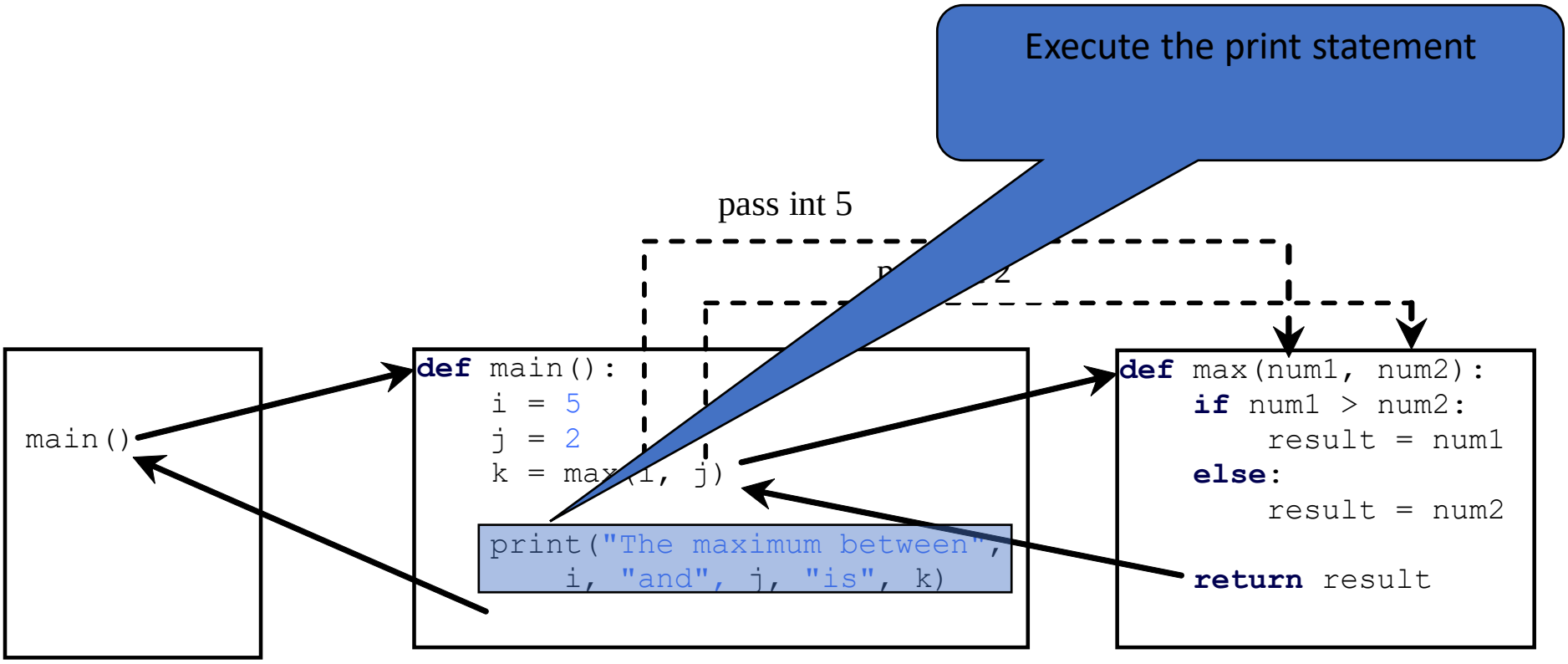
Trace Function Invocation



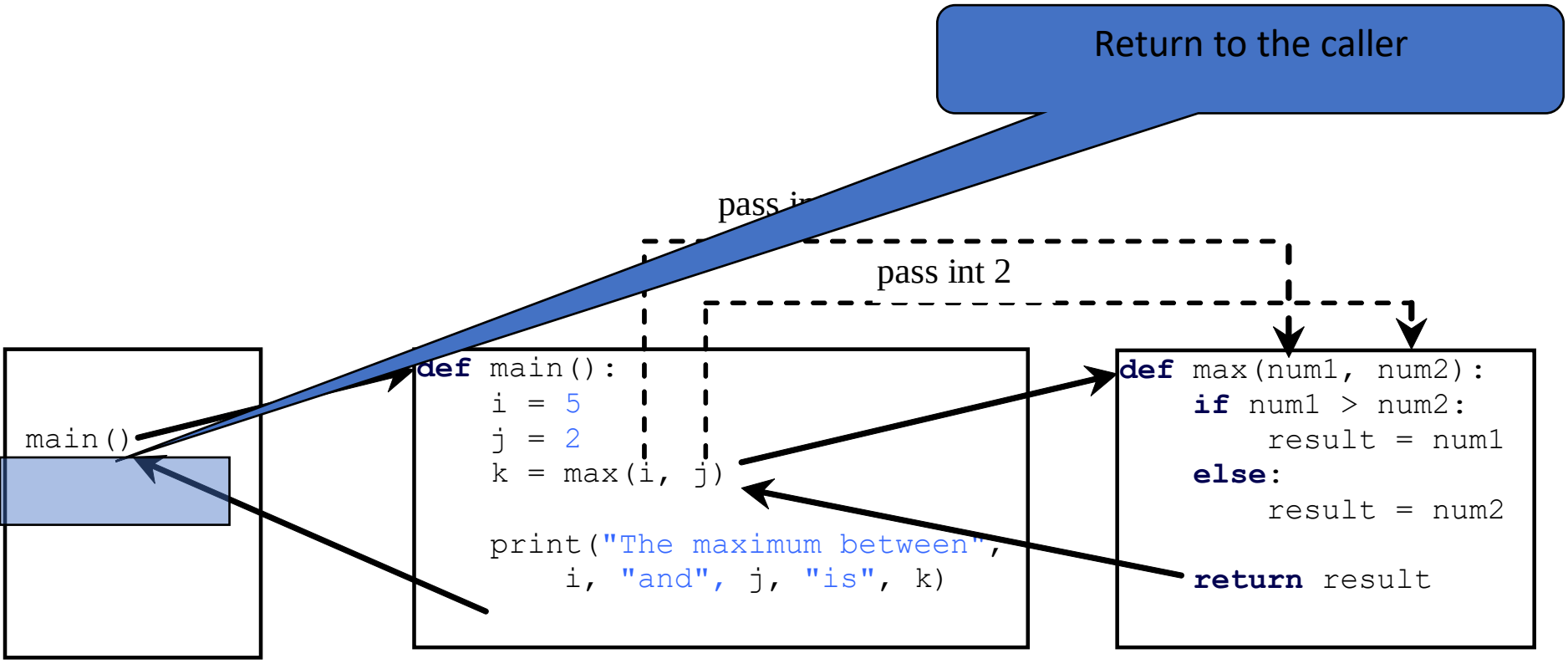
Trace Function Invocation



Trace Function Invocation



Trace Function Invocation



Pass by value and pass by reference

- In pass-by-value, the function receives a copy of the argument objects passed to it by the caller, stored in a new location in memory.
- You pass values of parameter to the function, if any kind of change done to those parameters inside the function ,those changes not reflected back in your actual parameters.
- In pass-by-reference, the function receives reference to the argument objects passed to it by the caller, both pointing to the same memory location.
- You pass reference of parameters to your function, if any changes made to those parameters inside function those changes are get reflected back to your actual parameters.
- In Python, neither of these two concepts are applicable, rather the **values are sent to functions by means of object reference.**

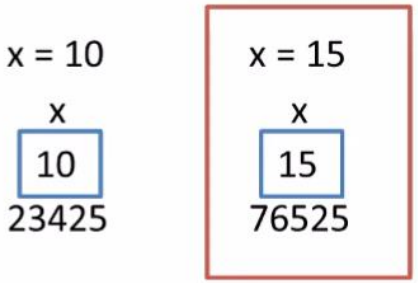
Pass-by-object-reference

- In Python, (almost) everything is an object. What we commonly refer to as “variables” in Python are more properly called names. Likewise, “assignment” is really the binding of a name to an object. Each binding has a scope that defines its visibility, usually the block in which the name originates.
- In Python, values are passed to function by object reference.
- if object is immutable (not modifiable) than the modified value is not available outside the function.
- if object is mutable (modifiable) than modified value is available outside the function.

Pass-by-object-reference

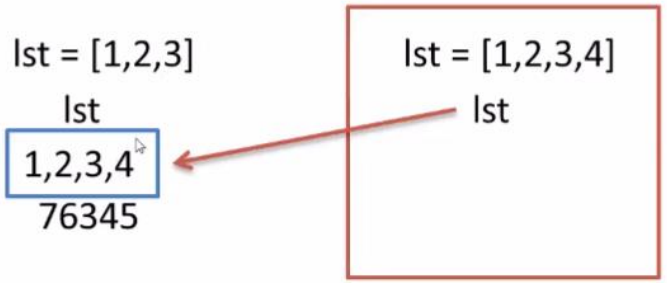
- **Immutable objects:** int, float, complex, string, tuple, frozen set, bytes.
- **Mutable objects**: list, dict, set, byte array

```
def val(x):
    x = 15
    print(x, id(x))

x = 10
val(x)
print(x, id(x))
```



A new object is created in the memory because integer objects are immutable (not modifiable).

```
def val(lst):
    lst.append(4)
    print(lst, id(lst))

lst = [1, 2, 3]
print(lst, id(lst))
val(lst)
```



A new object is not created in the memory because list objects are mutable (modifiable). It simply add new element to the same object.

Modules and the import statement

- A program can load a module file by using the import statement
- Use the name of the module
- Functions and variables names in the module must be qualified with the module name. e.g. math.pi

```
import math
radius = 2.0
area = math.pi * (radius ** 2)


math.log(5)
1.6094379124341003

import sys
sys.exit()
```

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 )
```

formal parameter

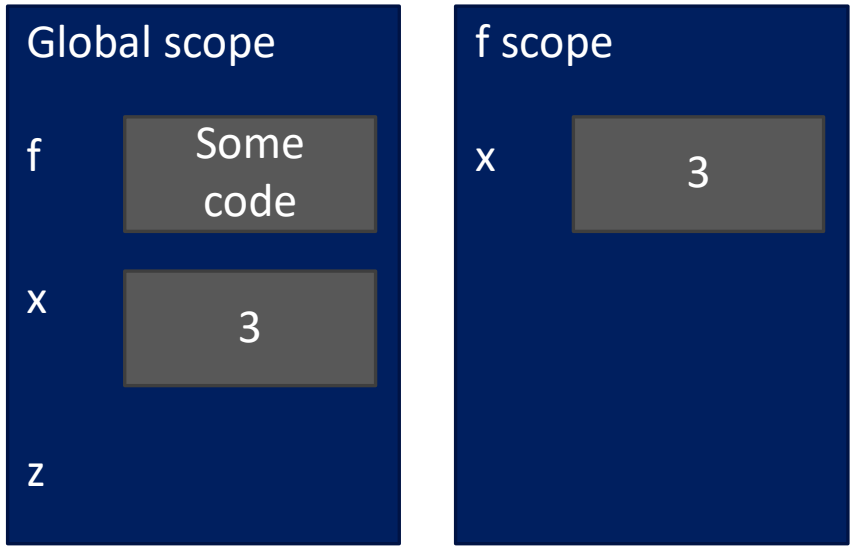
actual parameter

Function definition

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

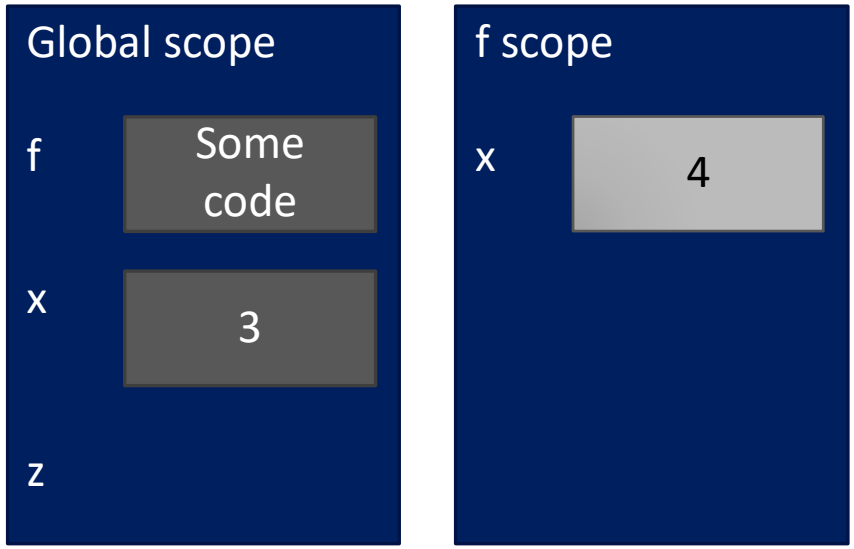
VARIABLE SCOPE

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

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



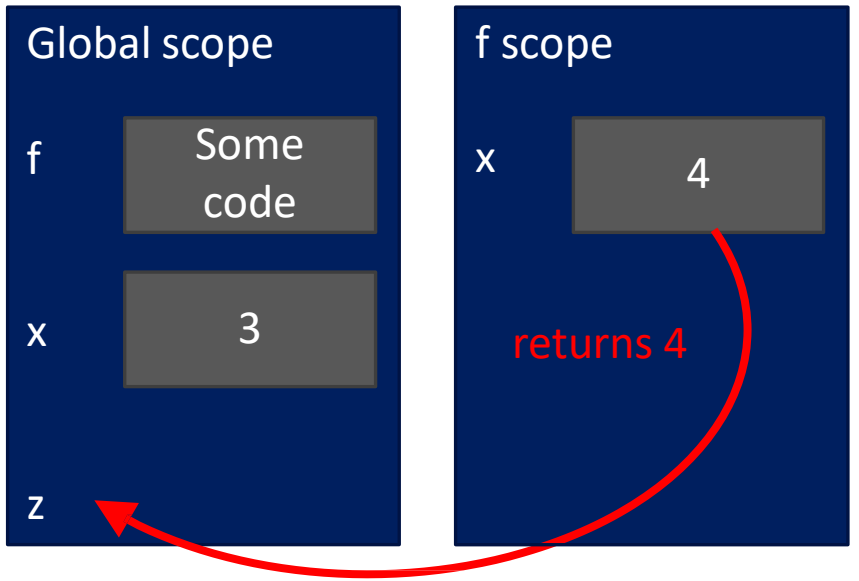
VARIABLE SCOPE

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

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



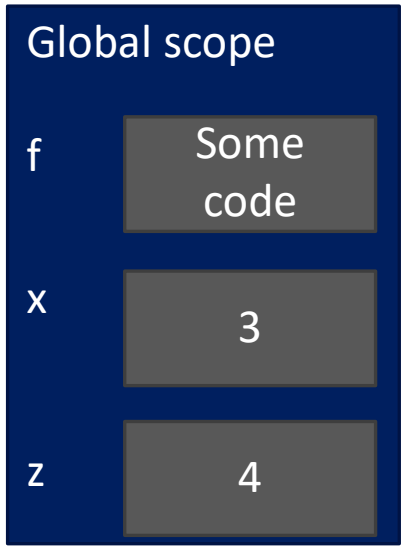
VARIABLE SCOPE

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

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



VARIABLE SCOPE

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

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



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)
```

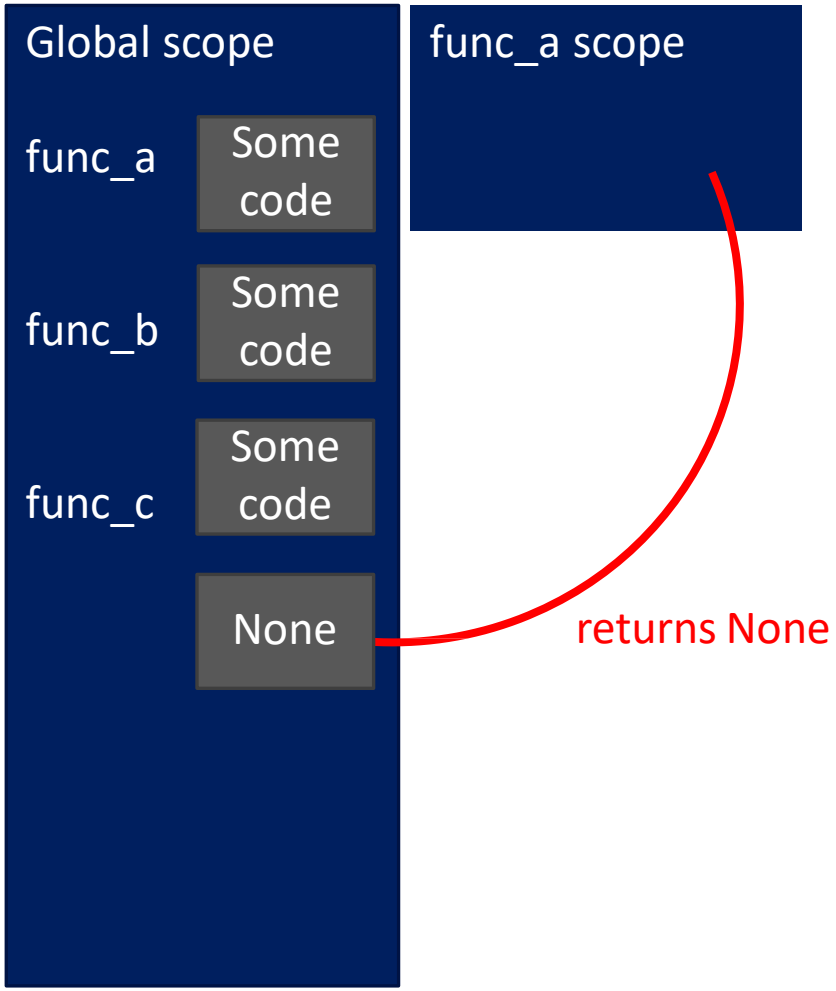
call `func_a`, takes no parameters

call `func_b`, takes one parameter

call `func_c`, takes one parameter, another function

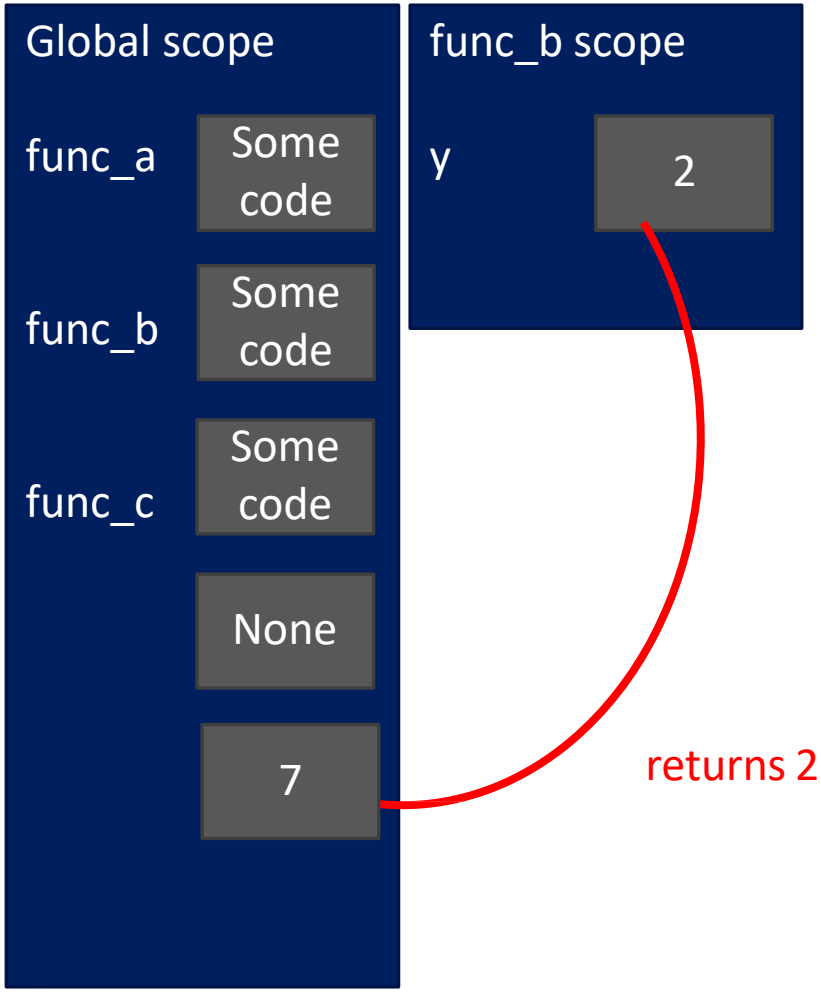
FUNCTIONS AS ARGUMENTS

```
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)
```



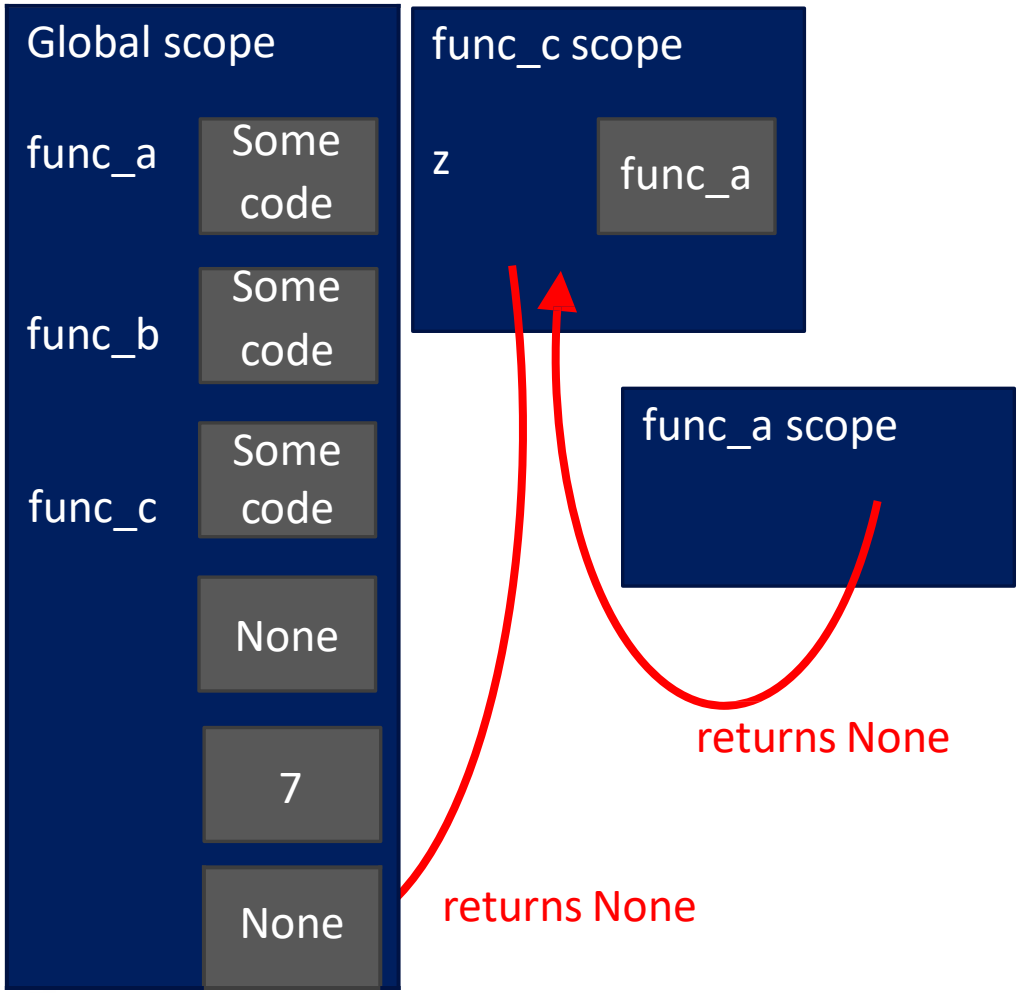
FUNCTIONS AS ARGUMENTS

```
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

```
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 += 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)


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

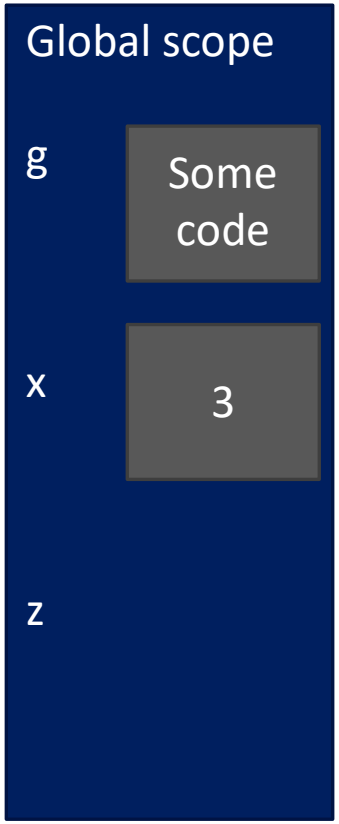
Python Tutor is your best friend to help sort this out!

http://www.pythontutor.com/

SCOPE DETAILS

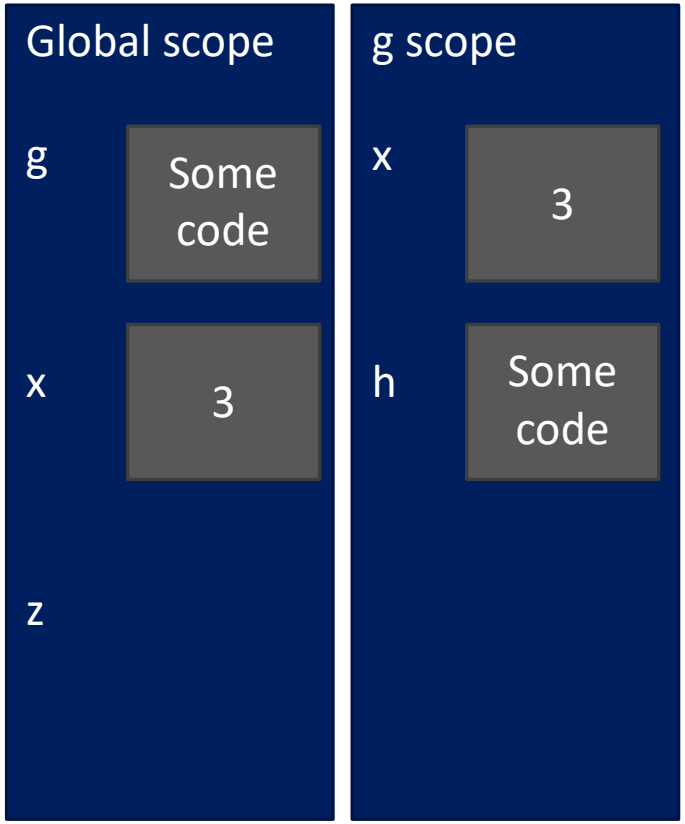
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

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

Some code



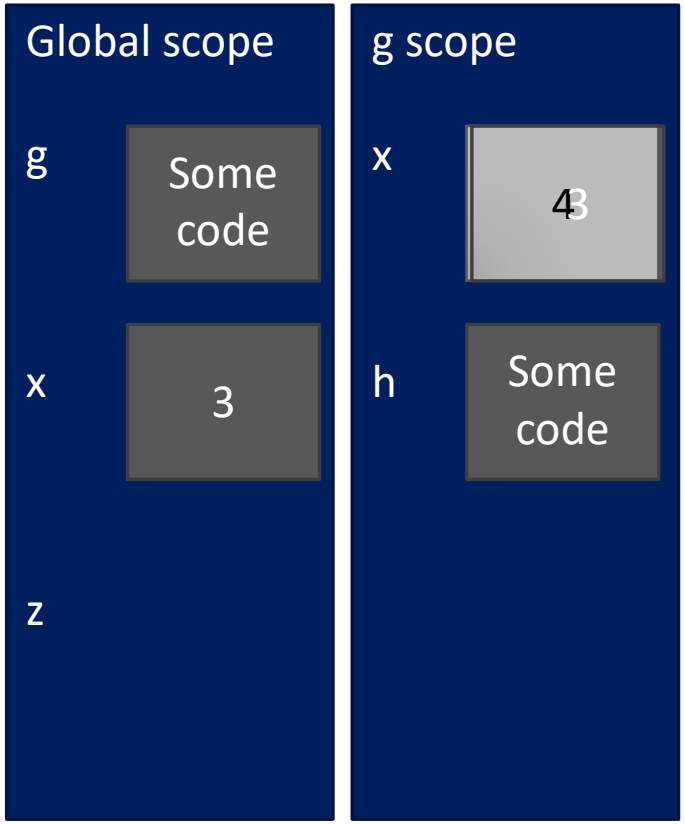
SCOPE DETAILS

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

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



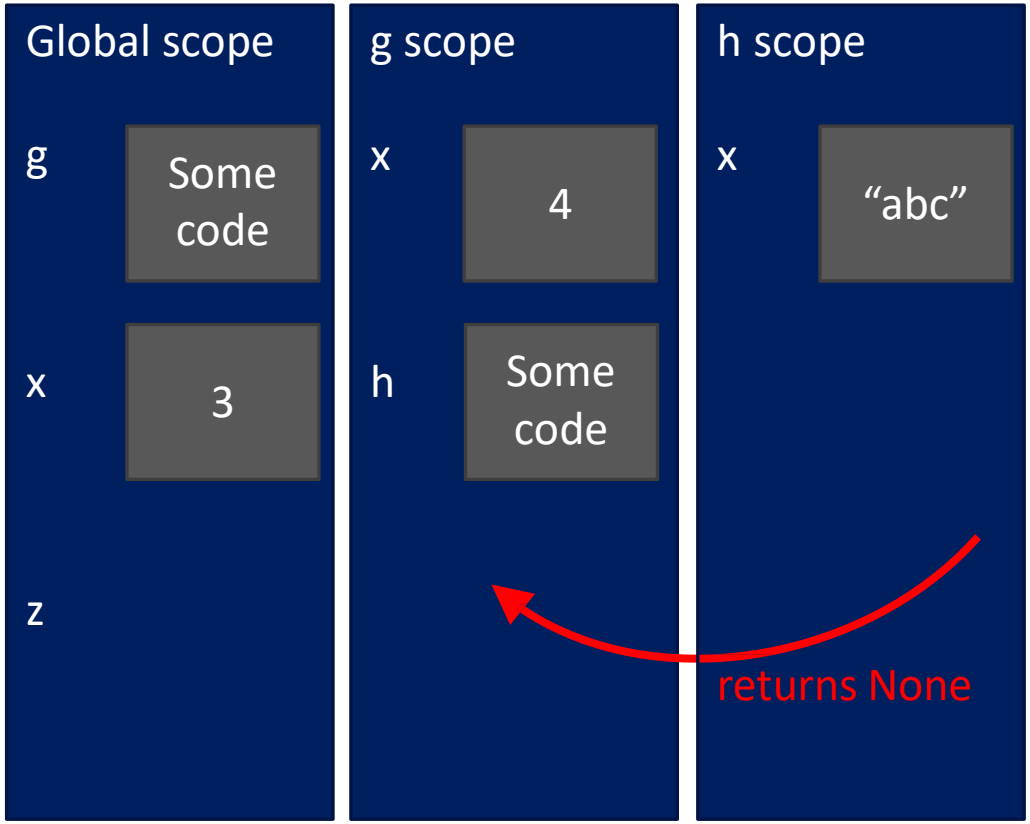
SCOPE DETAILS

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

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



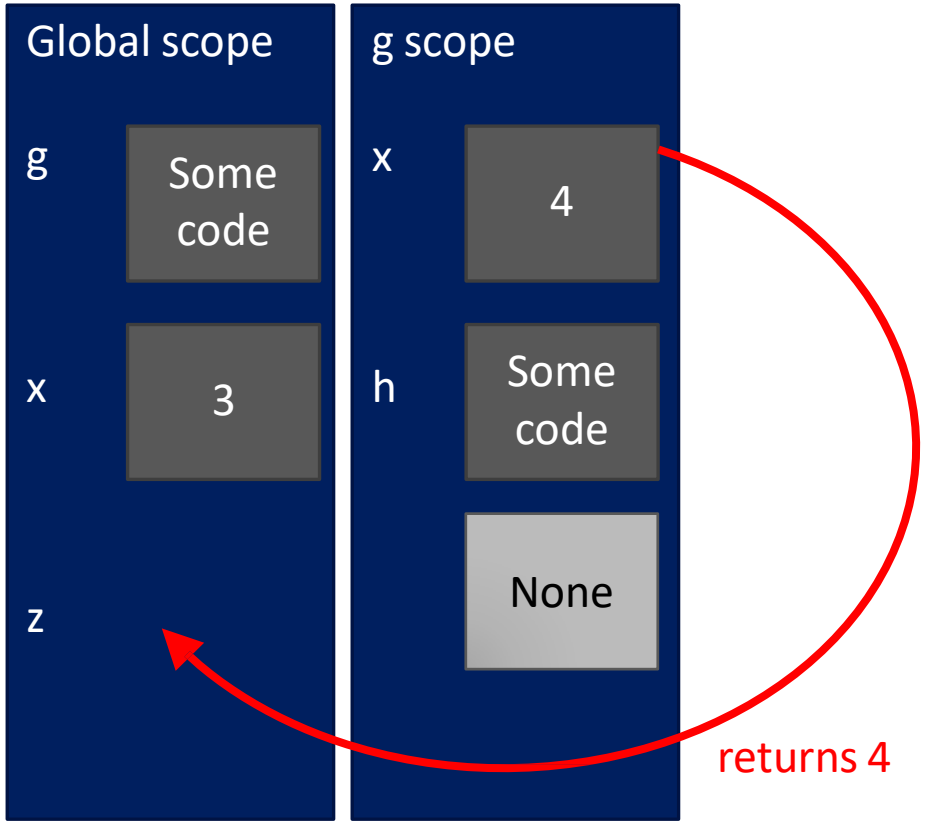
SCOPE DETAILS

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

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



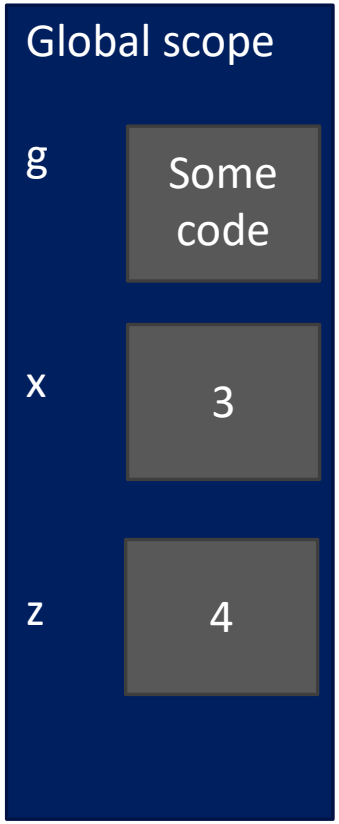
SCOPE DETAILS

```
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)
```



Default Parameter Value

- If we call the function without argument, it uses the default value

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Default Parameter Value

- A function can have multiple parameters with default values. Parameters with default values need to be grouped into the last (or unique) parameters of a function. When calling a function with multiple parameters with default values, we can only omit the parameters in the order from right to left and must drop them consecutively.

```
def func(a, b = 3, c = 5):
    return a + b + c
print(func(1), func(1, 5), func(1, 5, 9))
```

Lambda function

- A lambda function is a small anonymous function.
- In contrast to a normal function, a Python lambda function is a single expression. Although, in the body of a lambda, you can spread the expression over several lines using parentheses or a multiline string, it remains a single expression
- Syntax:

```
lambda arguments : expression
```

Lambda function

```
x = lambda a : a + 10
print(x(5))


x = lambda a, b : a * b
print(x(5, 6))


x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Lambda function

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

Lambda function

```
(lambda x: x + 1)(2)


(lambda x, y: x + y)(2, 3)


>>> high_ord_func = lambda x, func: x + func(x)
>>> high_ord_func(2, lambda x: x * x)
6
>>> high_ord_func(2, lambda x: x + 3)
7
```

Recursion

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

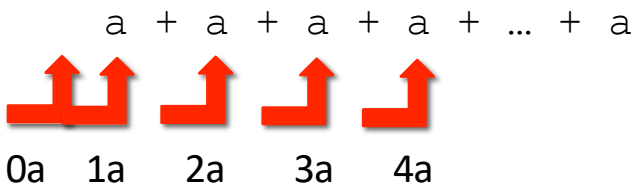
- **function calls itself**
  - in programming, goal is to NOT have infinite recursion
    - must have **1 or more base cases** that are easy to solve
    - must solve the same problem on **some other input** with the goal of simplifying the larger problem input

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
```

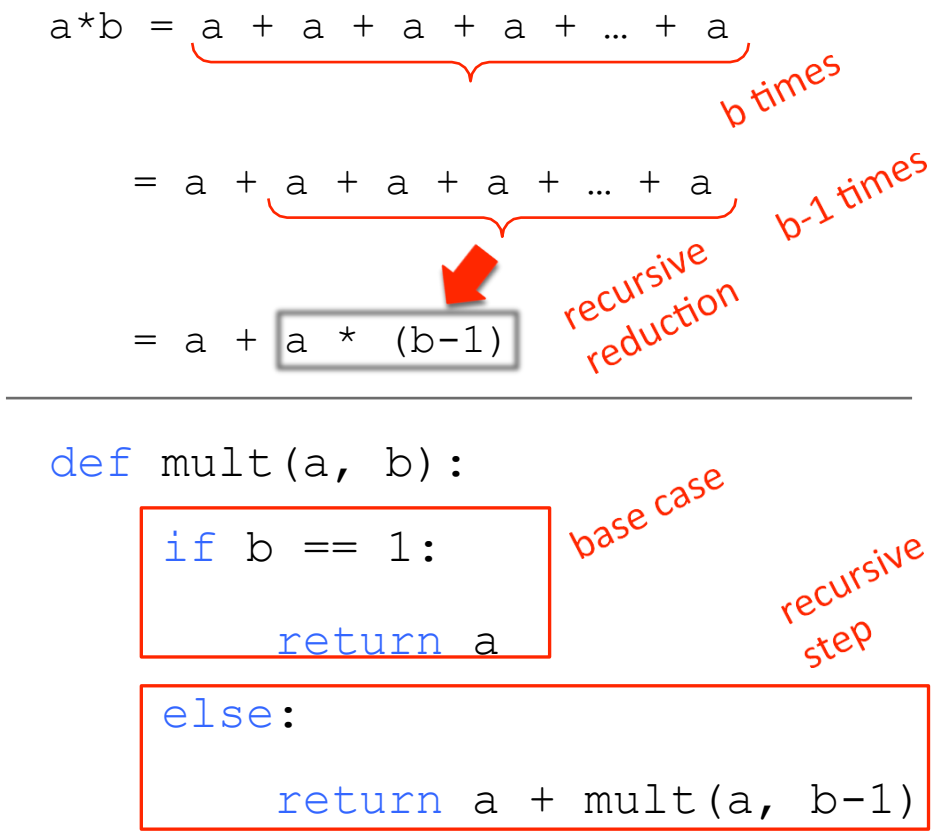
iteration

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



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

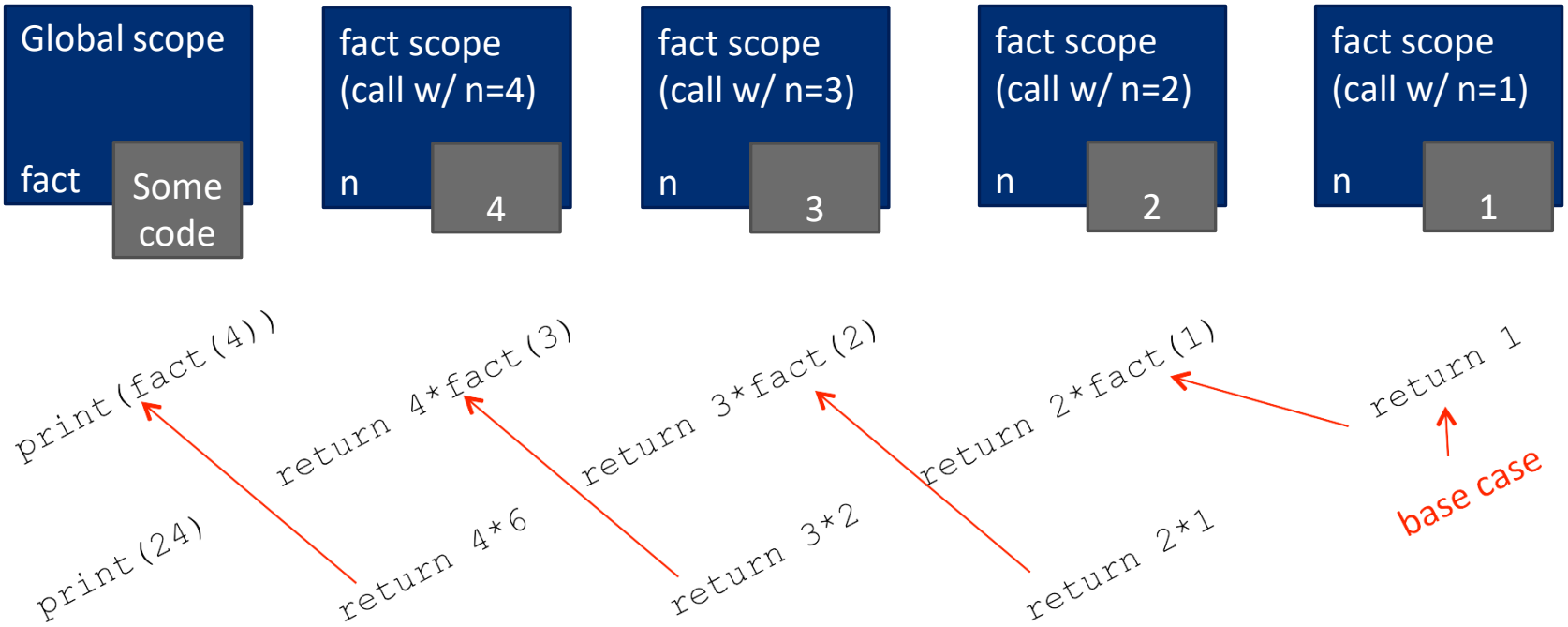
- how to reduce problem? Rewrite in terms of something simpler to reach base case

  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

- 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

```
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

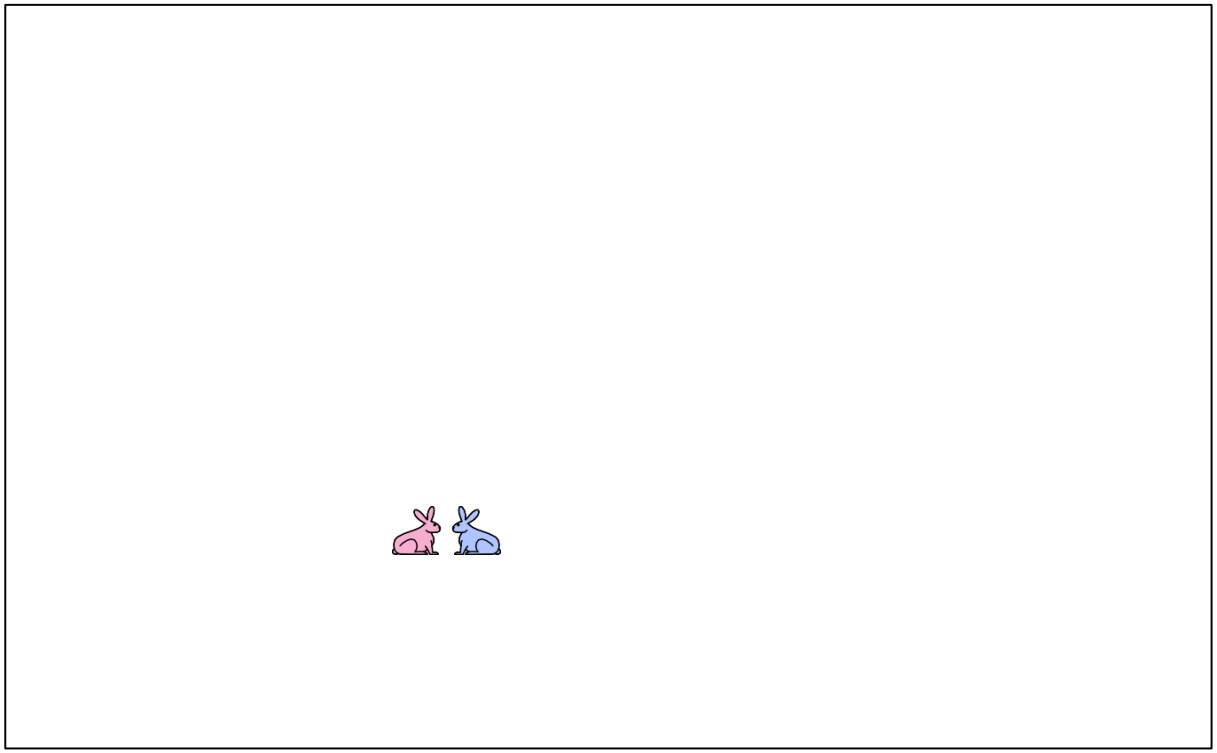
TOWERS OF HANOI

- 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

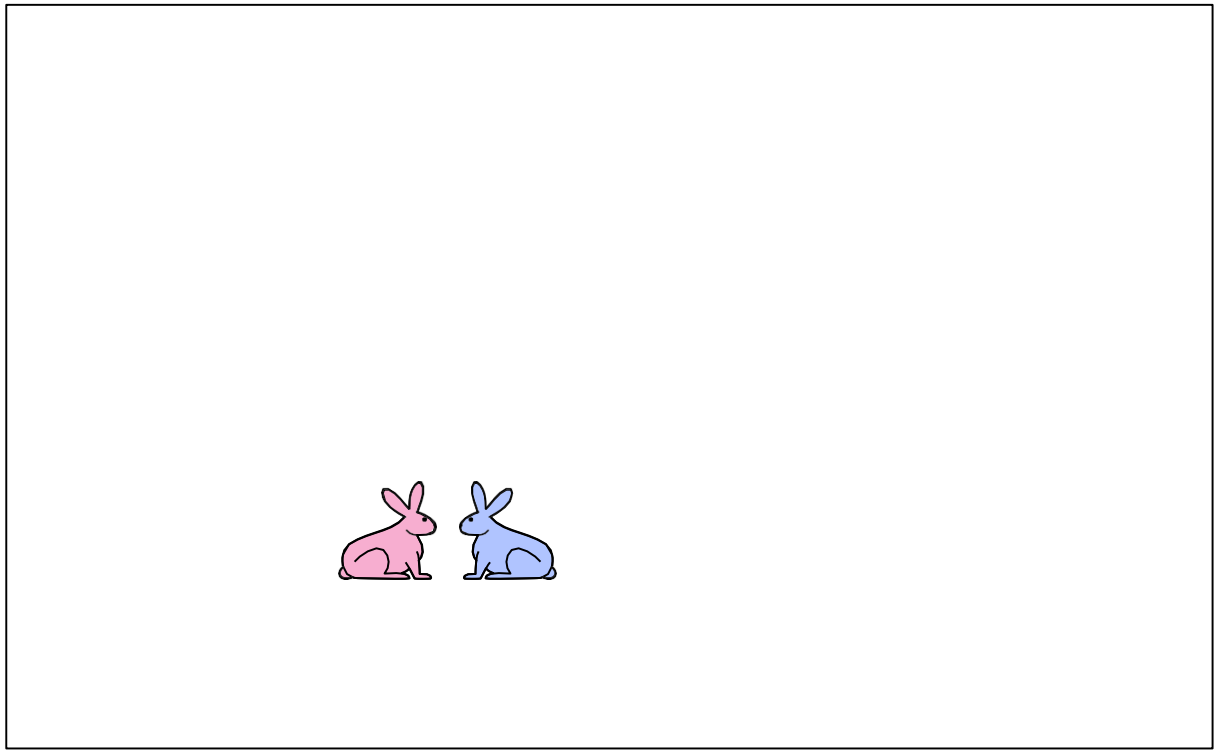
```
def printMove(fr, to):
    print('move from ' + str(fr) + ' to ' + str(to))

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

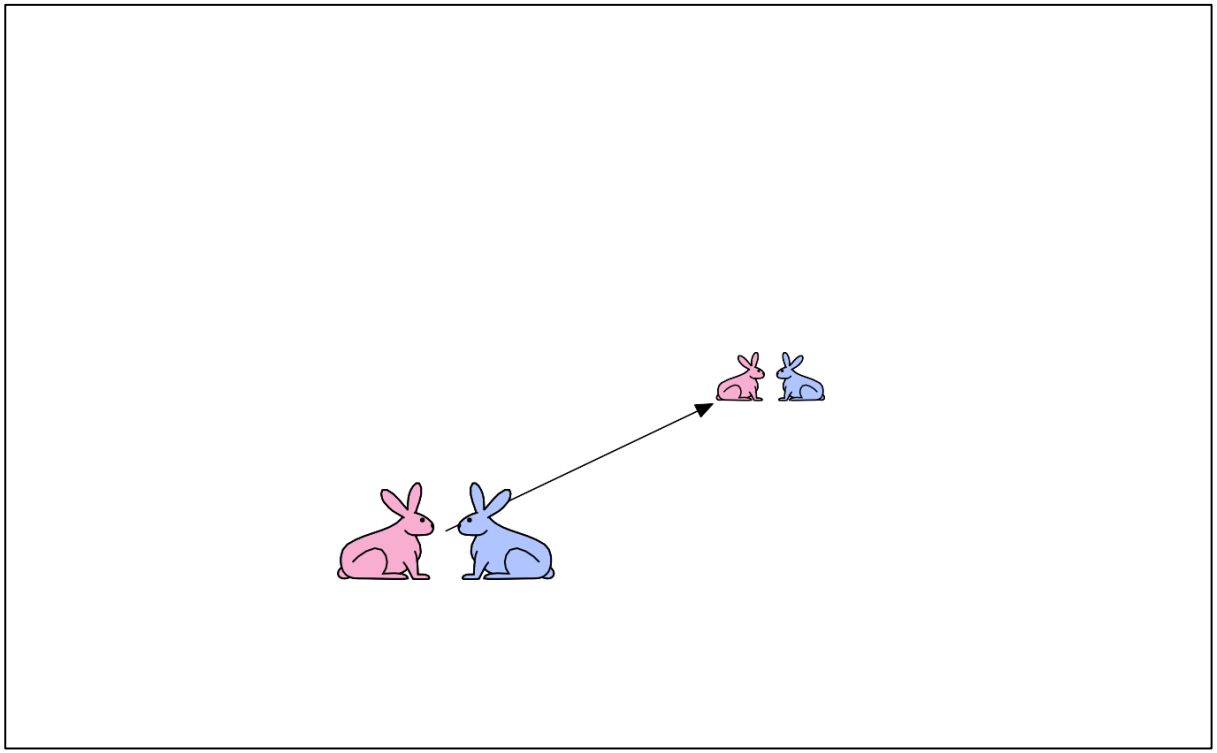
- Fibonacci numbers
  - Leonardo of Pisa (aka Fibonacci) modeled the following challenge
    - Newborn pair of rabbits (one female, one male) are put in a pen
    - Rabbits mate at age of one month
    - Rabbits have one month gestation period
    - Assume rabbits never die, that female always produces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



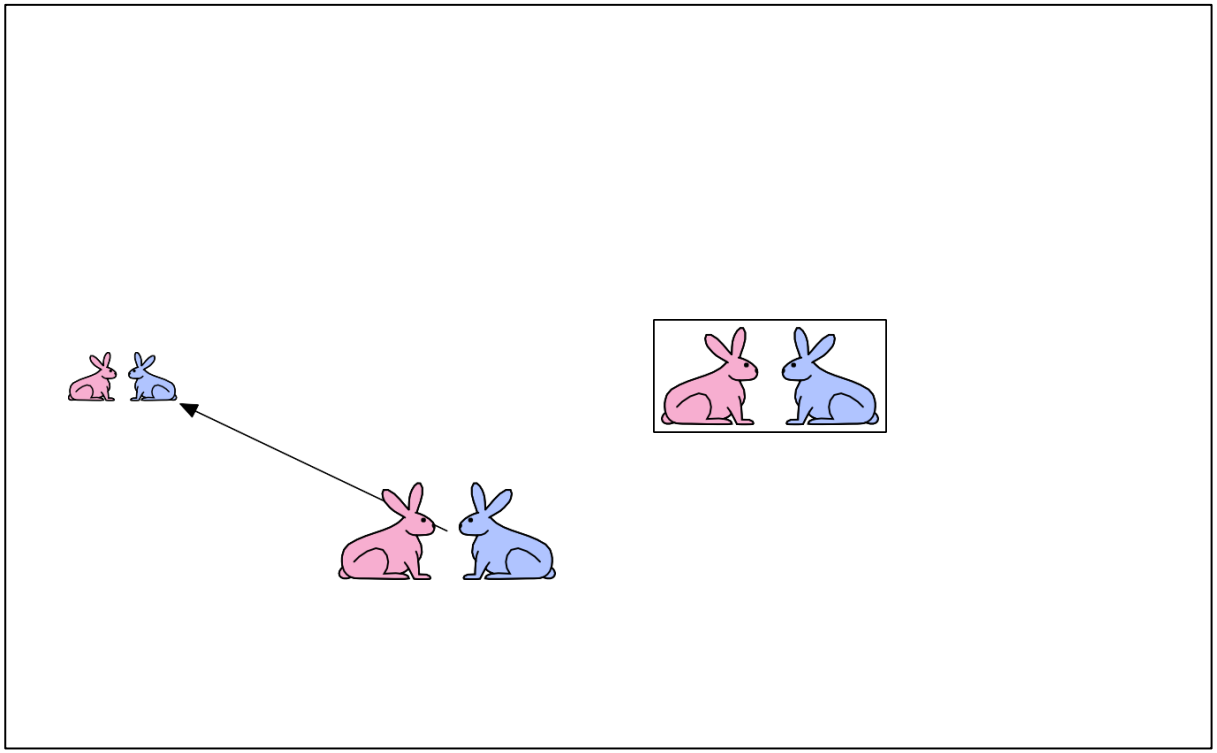
Demo courtesy of Prof. Denny Freeman and Adam Hartz



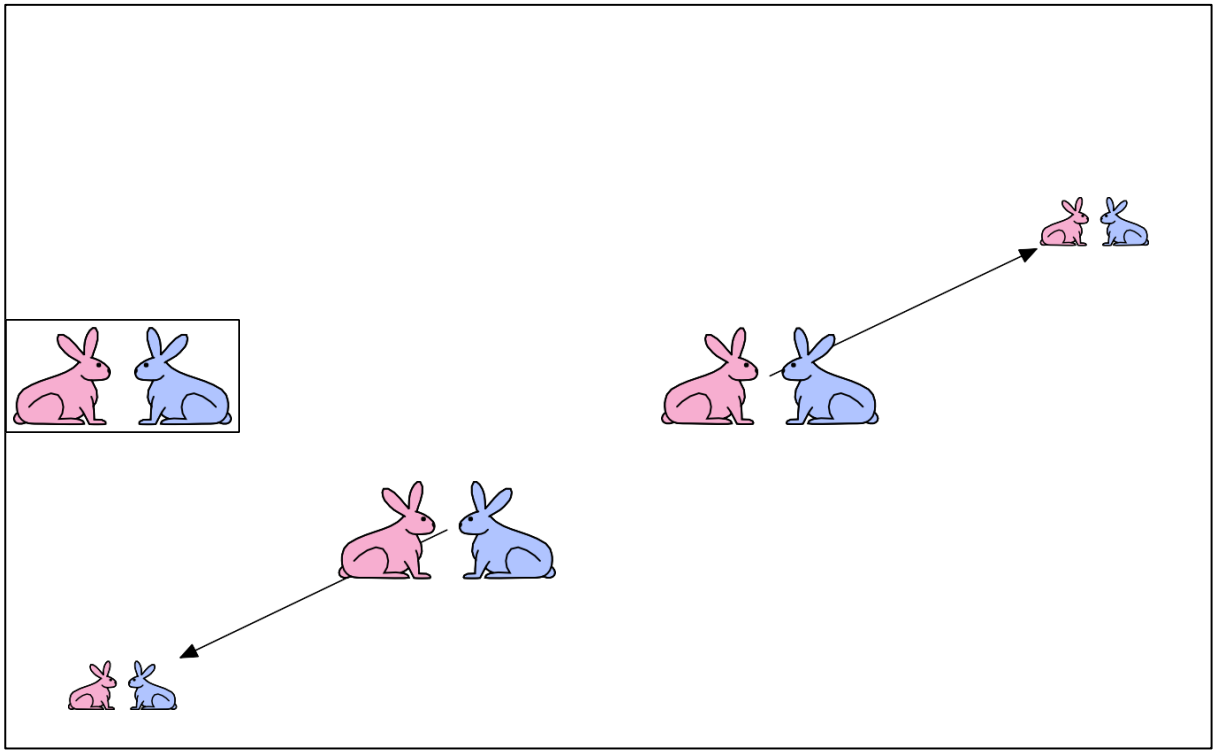
Demo courtesy of Prof. Denny Freeman and Adam Hartz



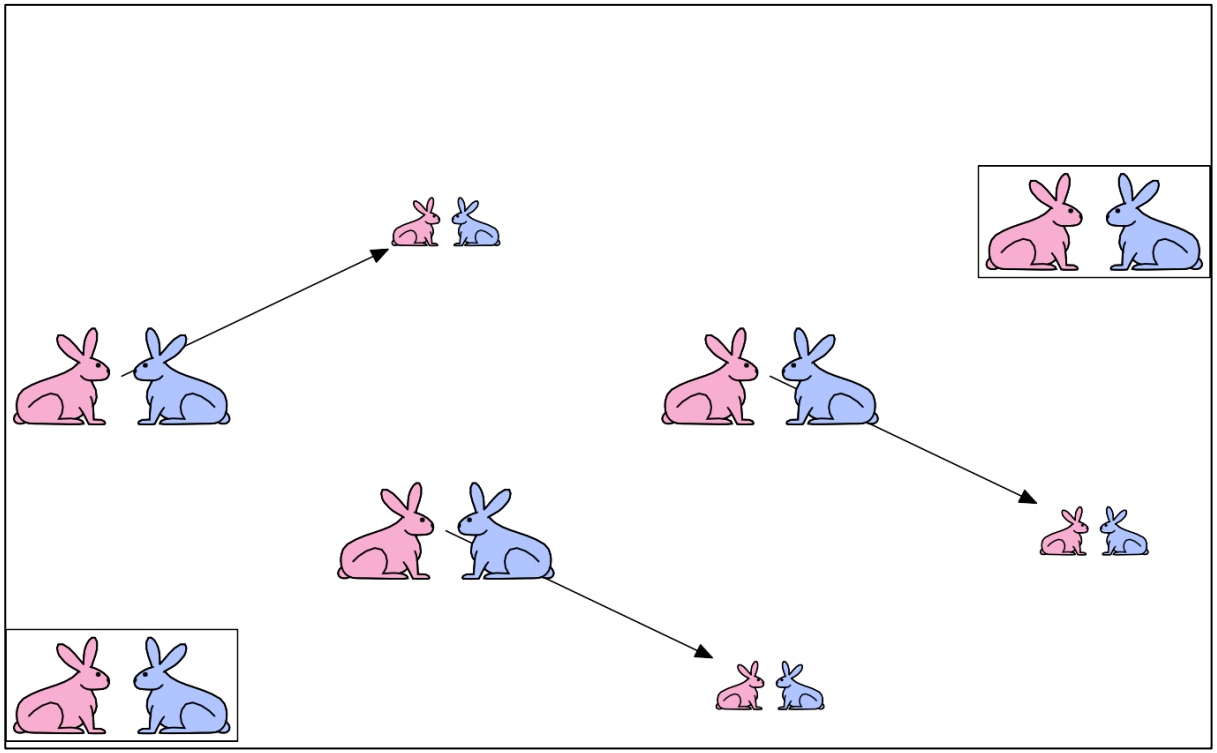
Demo courtesy of Prof. Denny Freeman and Adam Hartz



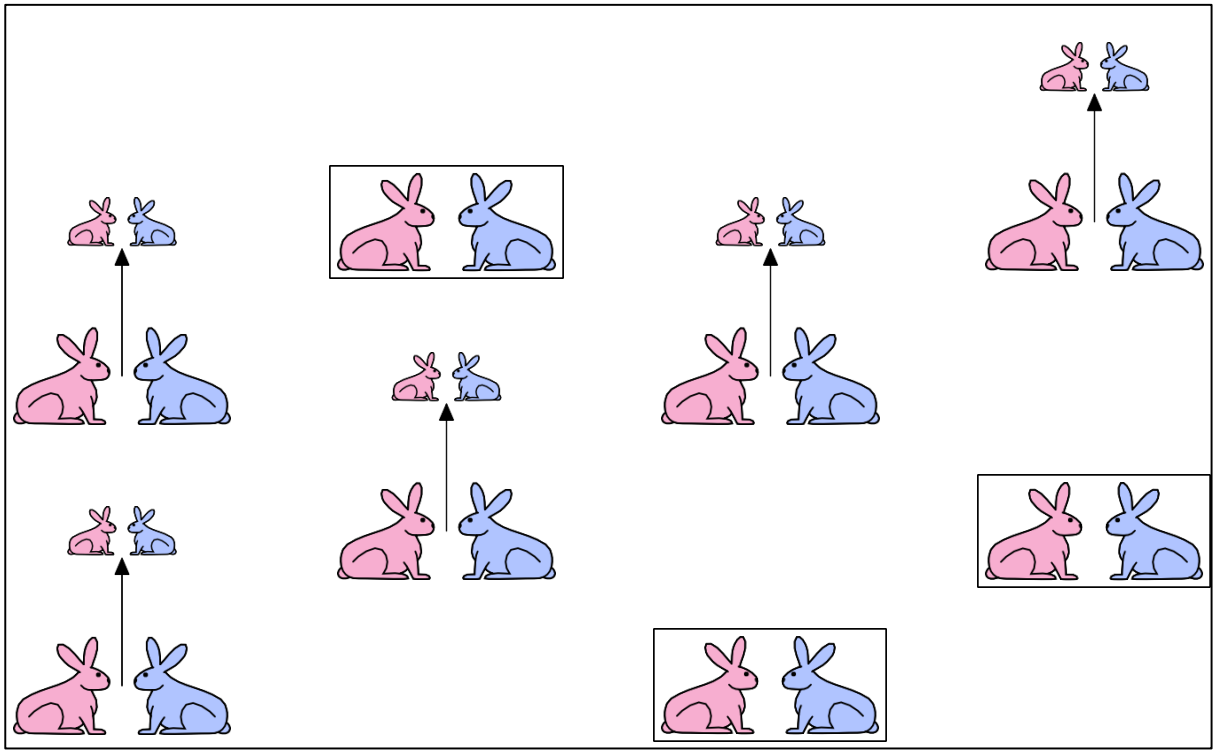
Demo courtesy of Prof. Denny Freeman and Adam Hartz



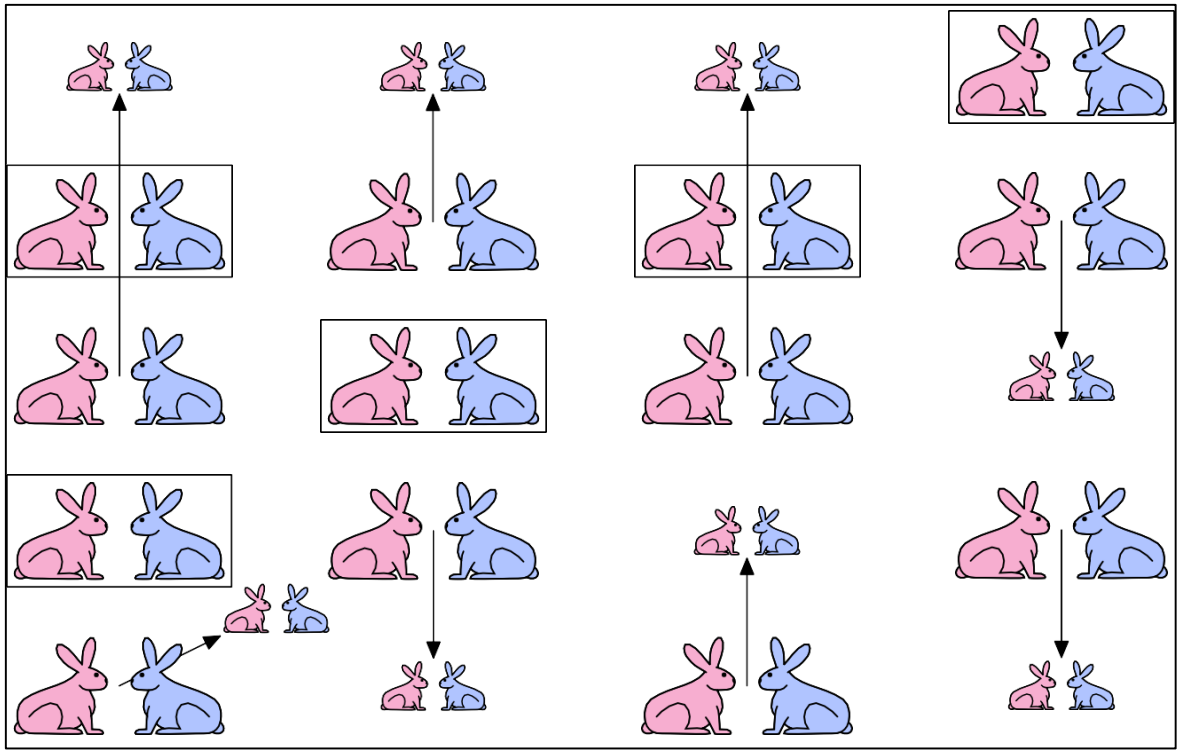
Demo courtesy of Prof. Denny Freeman and Adam Hartz

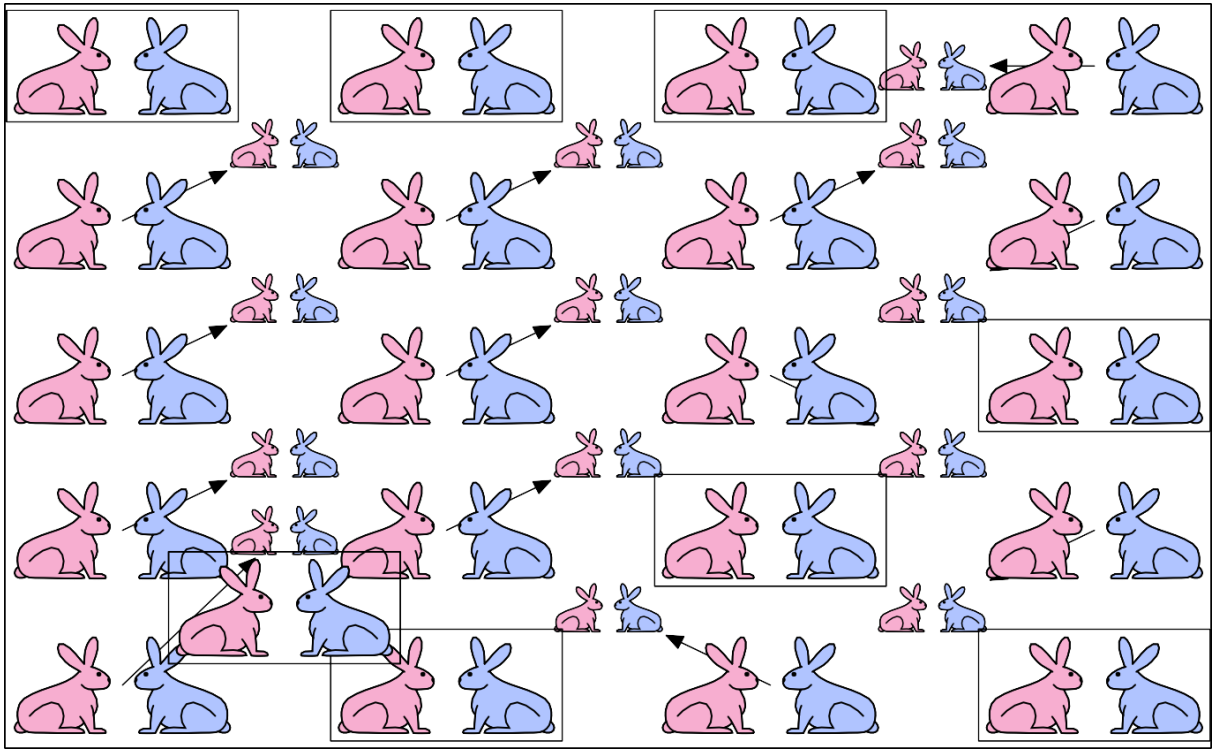


Demo courtesy of Prof. Denny Freeman and Adam Hartz

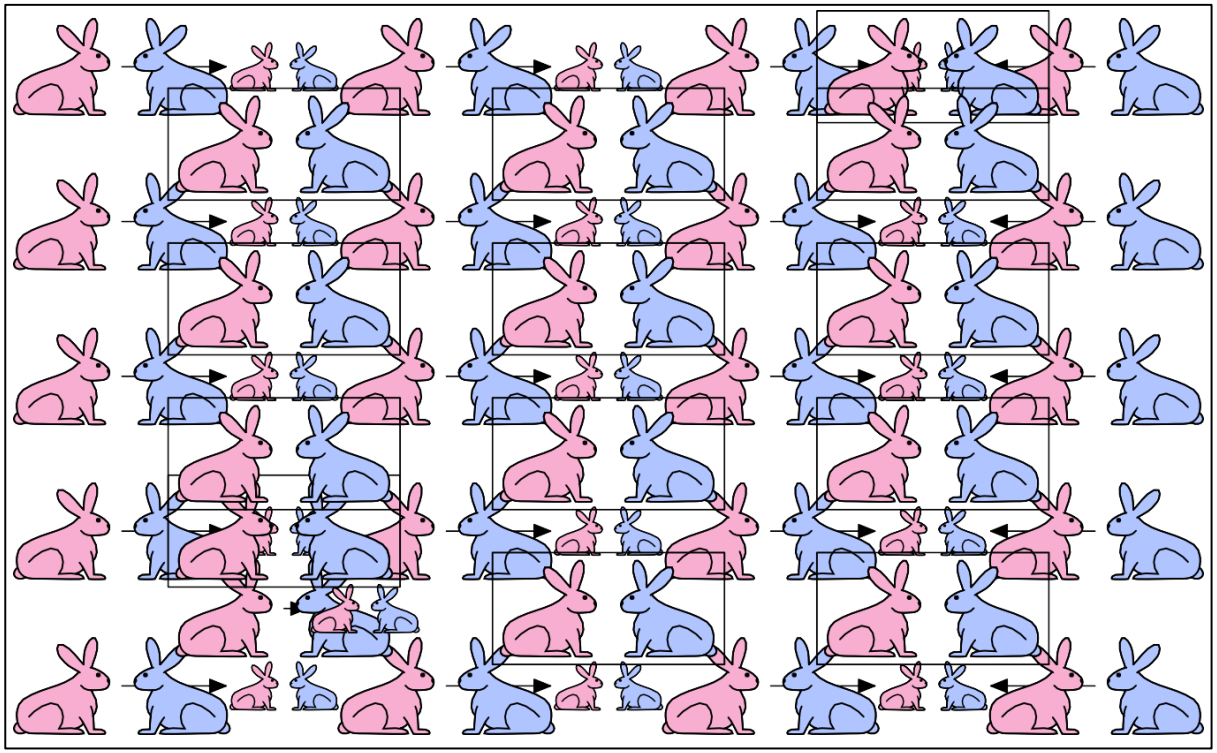


Demo courtesy of Prof. Denny Freeman and Adam Hartz





Demo courtesy of Prof. Denny Freeman and Adam Hartz



Demo courtesy of Prof. Denny Freeman and Adam Hartz

FIBONACCI

- Ayer one month (call it 0) – 1 female
- Ayer second month – still 1 female (now pregnant)
- Ayer 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))
```