

DECOMPOSITION, ABSTRACTION, FUNCTIONS

(download slides and .py files ••• follow along!)

6.0001 LECTURE 4

LAST TIME

- while loops vs for loops
- should know how to write both kinds
- should know when to use them
- guess-and-check and approximation methods
- bisection method to speed up programs

TODAY

- structuring programs and hiding details
- functions
- specifications
- keywords: `return` vs `print`
- scope

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

- a projector is a black box
- don't know how it works
- know the interface: input/output
- connect any electronic 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 CONCEPTS

TO PROGRAMMING!

CREATE STRUCTURE with DECOMPOSITION

- in projector 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 projector example, instructions for how to use it are sufficient, no need to know how to build one
- 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**

FUNCTIONS^{**}.

- write reusable pieces/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** Ex) `f(x) = def f(x):`
 - has **parameters** (0 or more)
 - has a **docstring** (optional but recommended)
 - has a **body**
 - **returns** something

HOW TO WRITE and CALL/INVOKE A FUNCTION

The diagram illustrates the structure of a Python function definition and its invocation. It features a red-bordered box containing the code, with various parts labeled in red and blue.

Labels:

- keyword**: Points to the word "def".
- name**: Points to the identifier "is_even".
- parameters or arguments**: Points to the parameter "i".
- specification docstring**: Points to the multi-line string at the top of the function body.
- body**: Points to the code block below the docstring.
- Parameter**: Points to the argument "3" in the call expression.

Annotations:

- A green arrow labeled "define function." points from the left towards the function definition.
- A blue oval labeled "do anything" is positioned over the "body" label.
- A red annotation on the right side states: "later in the code, you call the function using its name and values for parameters".

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

is_even(3)
```

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
```

keyword

expression to
evaluate and return

run some
commands

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) : formal parameter
 x = x + 1
 print('in f(x) : x = ', x)
 return x
body
keyword.

x = 3
z = f(x) actual parameter

not in f(x) scope.
⇒ *call function*

Function definition) *f(x) scope.*

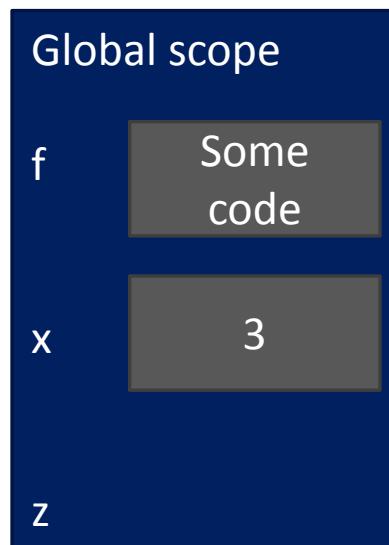
Main program code
* initializes a variable x
* makes a function call f(x)
* assigns return of function to variable z

VARIABLE SCOPE

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

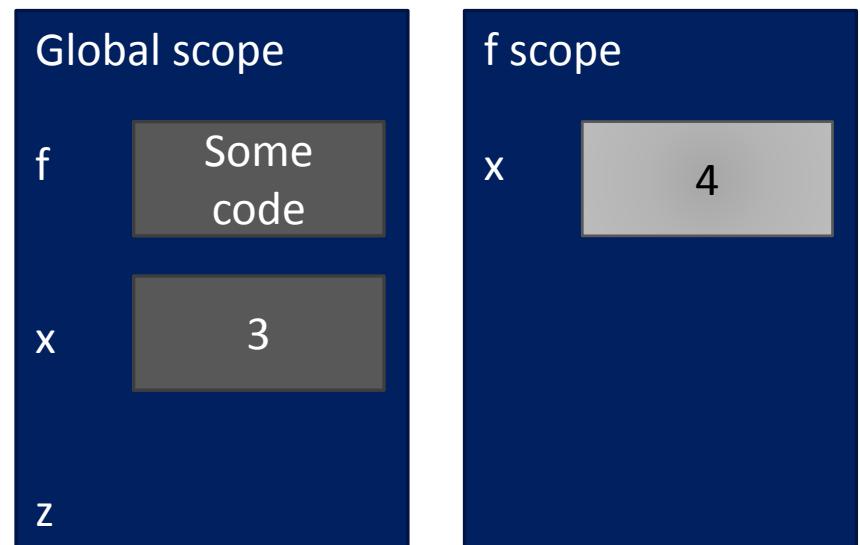
x = 3

z = f(x) *function call*



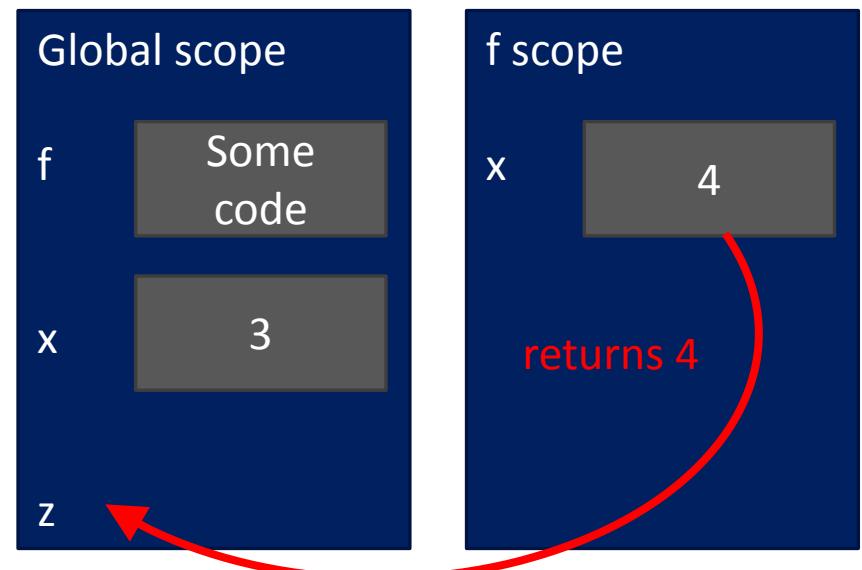
VARIABLE SCOPE

```
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

only function
return *only one*

- outside of function
- inside of function
print *~only one~*

- 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

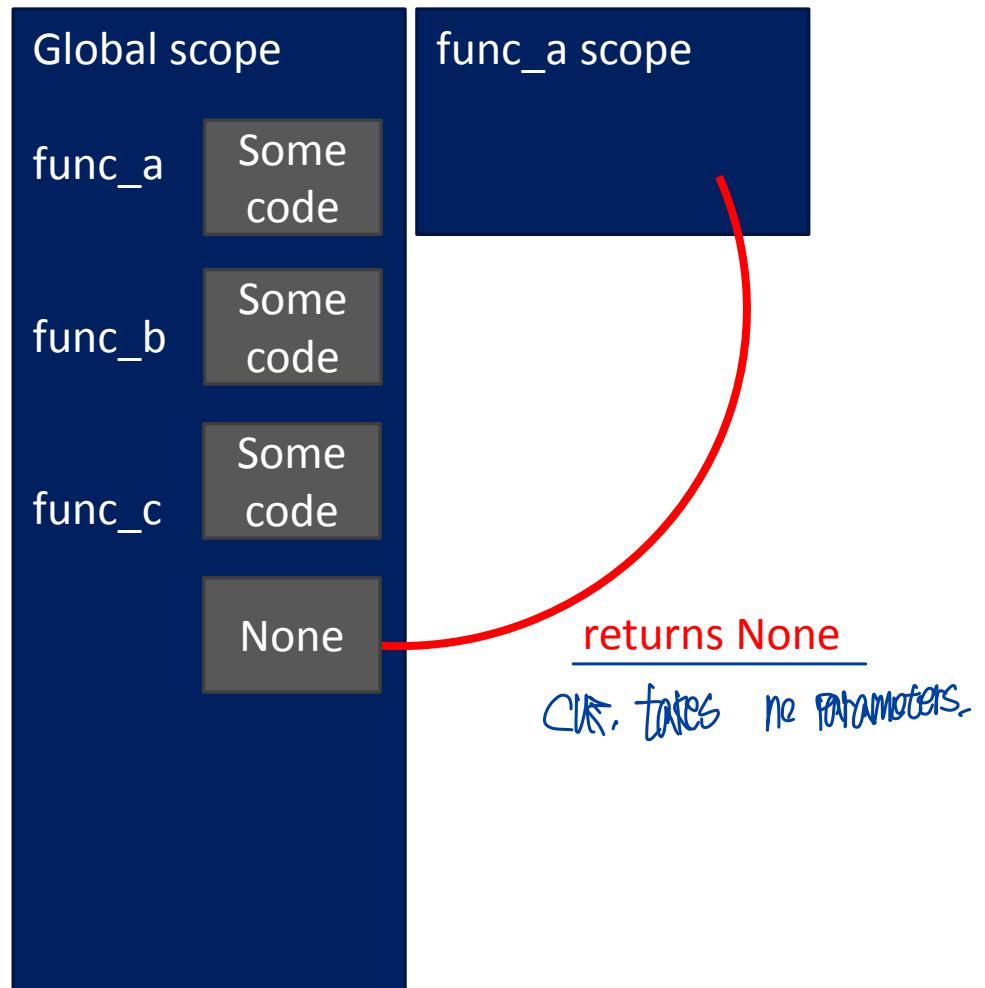
def func_c(z):
    print 'inside func_c'
    return z()

print func_a()          call function. no parameters
print 5 + func_b(2)     have parameters.  
parameter is "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
→func_a.

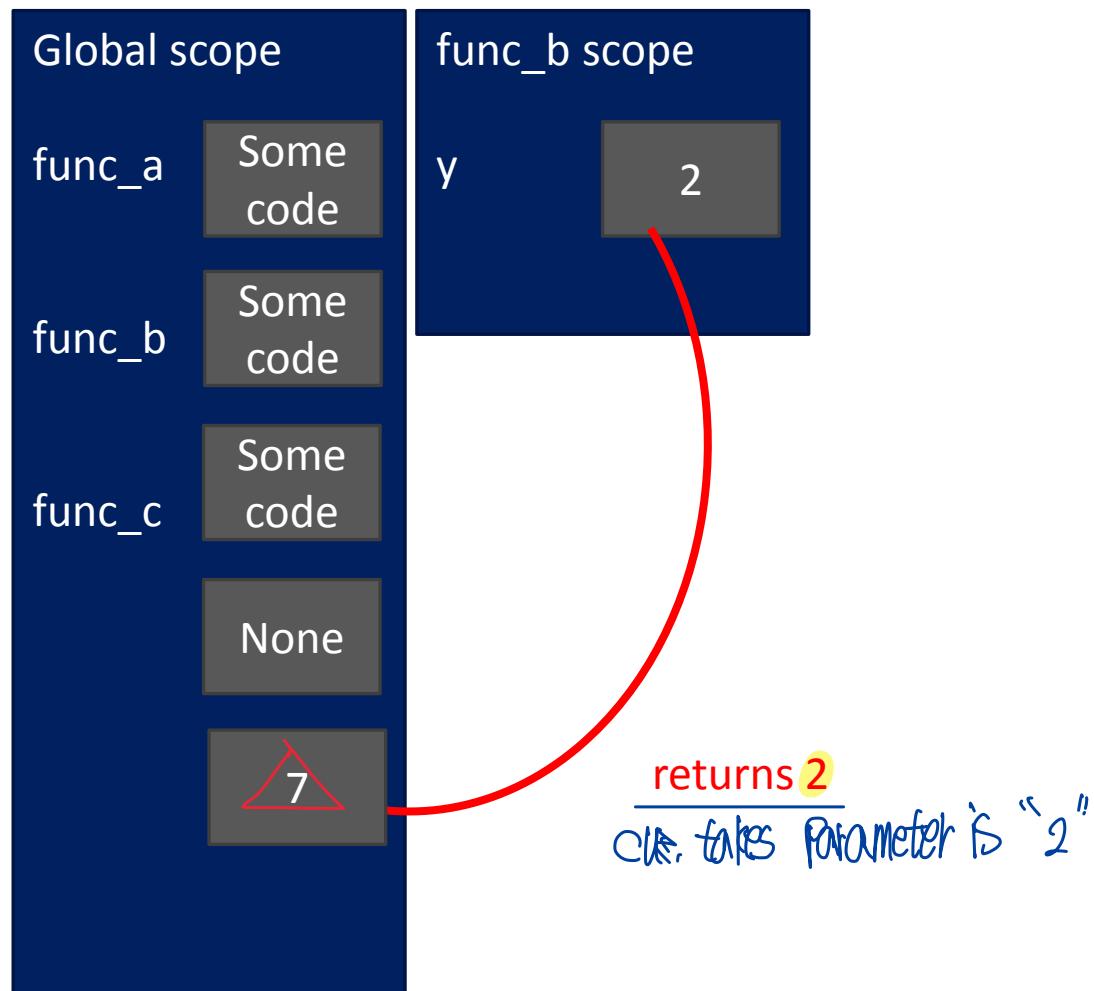
FUNCTIONS AS ARGUMENTS

```
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)
```



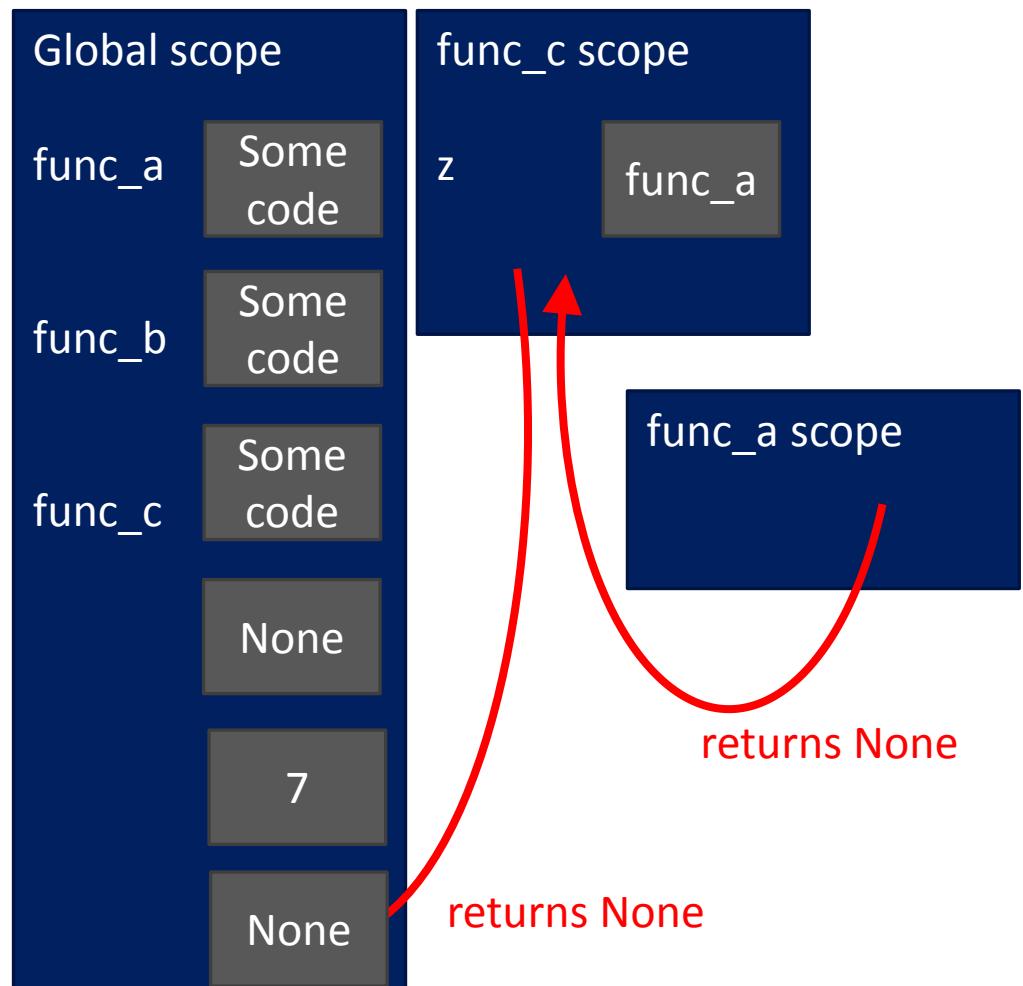
FUNCTIONS AS ARGUMENTS

```
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 use **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 use **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)
```

* from
global/main
program scope

HARDER SCOPE EXAMPLE



IMPORTANT
and
TRICKY!

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

<http://www.pythontutor.com/>

SCOPE DETAILS

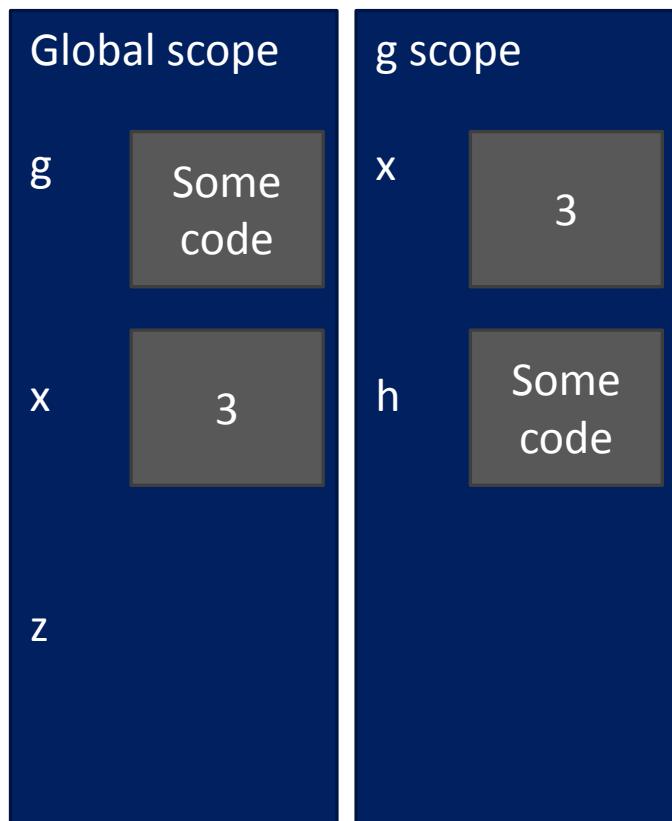
```
def g(x):  
    def h():  
        x = 'abc'  
        x = x + 1  
        print('g: x =', x)  
        h()  
    return x  
  
x = 3  
z = g(x)
```

Some code



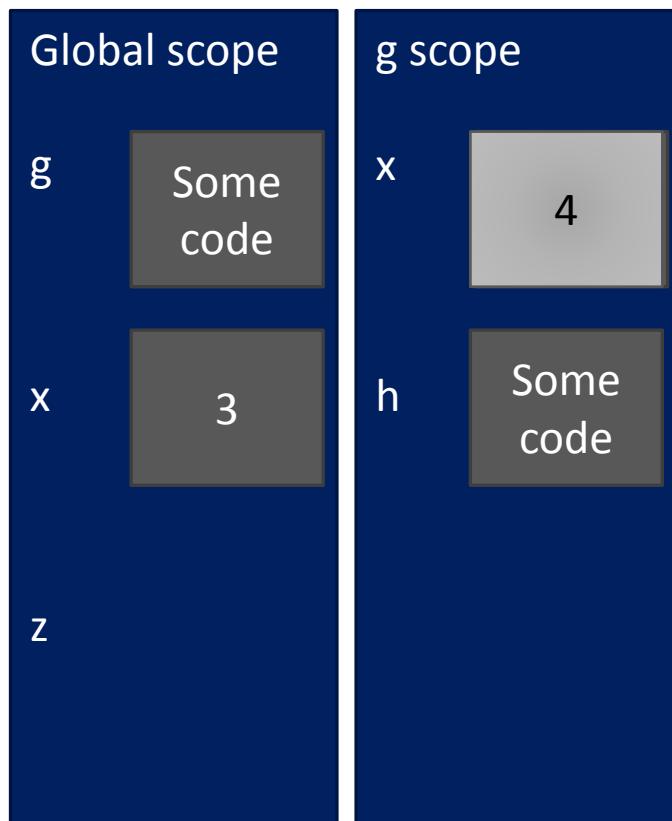
SCOPE DETAILS

```
def g(x):  
    def h():  
        x = 'abc'  
        x = x + 1  
        print('g: x =', x)  
        h()  
    return x  
  
x = 3  
z = g(x)
```



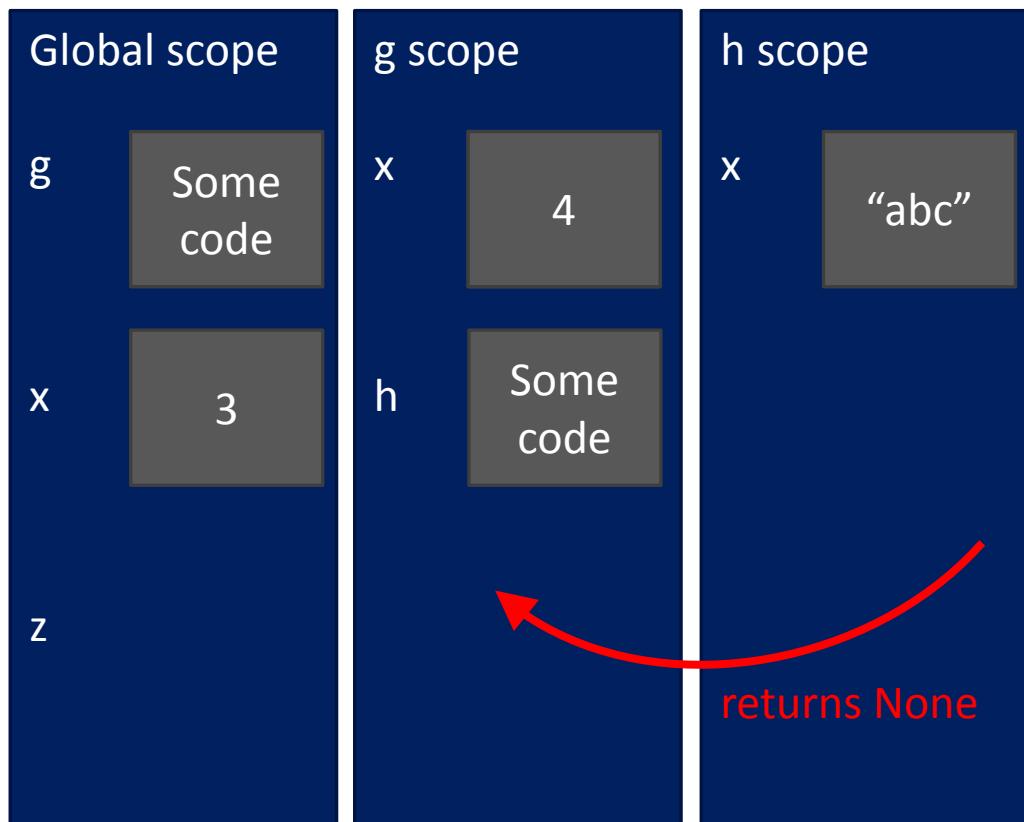
SCOPE DETAILS

```
def g(x):  
    def h():  
        x = 'abc'  
        x = x + 1  
        print('g: x =', x)  
        h()  
    return x  
  
x = 3  
z = g(x)
```



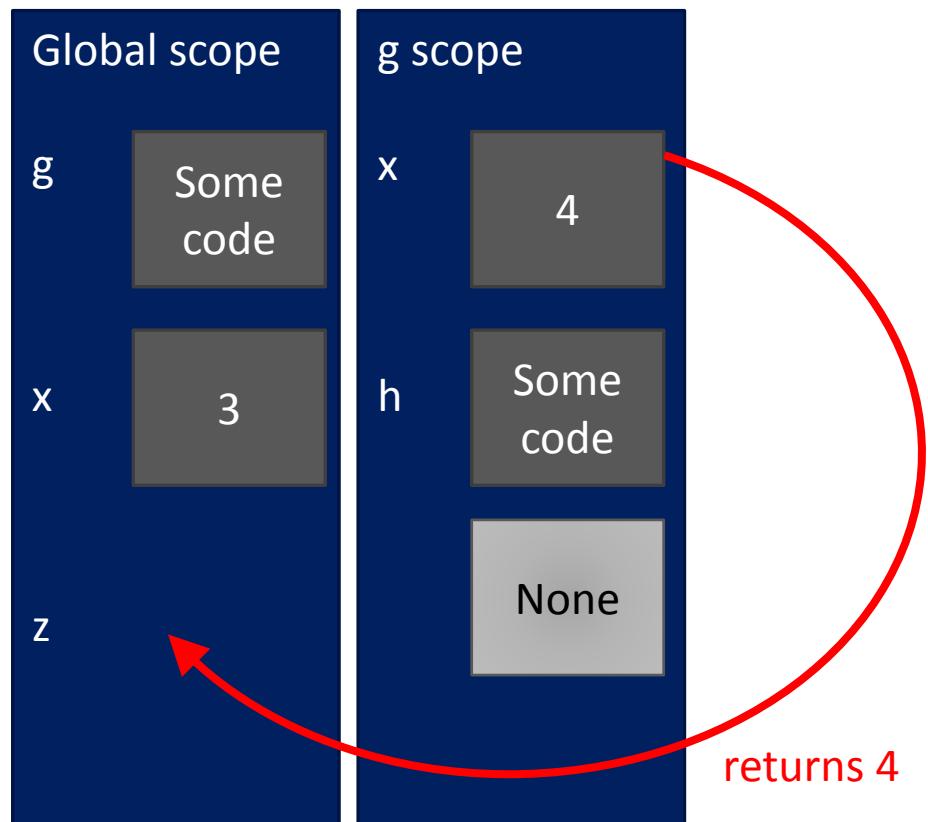
SCOPE DETAILS

```
def g(x):  
    def h():  
        x = 'abc'  
        x = x + 1  
        print('g: x =', x)  
        h()  
    return x  
  
x = 3  
z = g(x)
```



SCOPE DETAILS

```
def g(x):  
    def h():  
        x = 'abc'  
    x = x + 1  
    print('g: x =', x)  
    h()  
    return x  
  
x = 3  
z = g(x)
```



SCOPE DETAILS

```
def g(x):  
    def h():  
        x = 'abc'  
        x = x + 1  
        print('g: x =', x)  
        h()  
    return x  
  
x = 3  
z = g(x)
```



DECOMPOSITION & ABSTRACTION

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

MIT OpenCourseWare

<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python

Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>