

# DECOMPOSITION, ABSTRACTION, FUNCTIONS

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

COMP100 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 – ZOOM

---

- a zoom is a black box
- don't know how it works
- know the interface: input/output
- connect any user to a meeting  
and enable communication between different users
- black box somehow connects the user from input source to  
a meeting, enabling communication
- **ABSTRACTION IDEA:**  
do not need to know how zoom works to use it

# EXAMPLE – ZOOM

---

- connecting everyone for the class  
decomposed into separate tasks  
for separate users (student/instructor)
- each zoom takes input and produces separate output
- all zooms work together to produce the class
- **DECOMPOSITION IDEA:**  
different devices work together to achieve an end goal

# APPLY THESE CONCEPTS

---

# TO PROGRAMMING!



# CREATE STRUCTURE with DECOMPOSITION

---

- in zoom 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 zoom 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**
  - has **parameters** (0 or more)
  - has a **docstring** (optional but recommended)
  - has a **body**
  - **returns** something

# HOW TO WRITE and CALL/INVOKE A FUNCTION

---

keyword

```
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)
```

# HOW TO WRITE and CALL/INVOKE A FUNCTION

---

*keyword*  
*name*

```
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)
```

# HOW TO WRITE and CALL/INVOKE A FUNCTION

---

*keyword*  
`def`  
*name*  
`is_even`  
*parameters or arguments*  
`( 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)`

# HOW TO WRITE and CALL/INVOKE A FUNCTION

---

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

*keyword*  
*name*  
*parameters or arguments*  
*specification docstring*

```
is_even(3)
```

# HOW TO WRITE and CALL/INVOKE A FUNCTION

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

*keyword*

*name*

*parameters or arguments*

*specification docstring*

*body*

```
is_even(3)
```



# HOW TO WRITE and CALL/INVOKE A FUNCTION

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

is_even(3)
```

*keyword*

*name*

*parameters or arguments*

*specification, docstring*

*body*

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

*run some  
commands*

# 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*

*run some  
commands*

# 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

# 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
```

Function  
definition

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

*Function definition*

```
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
```

*formal  
parameter*

*Function  
definition*

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

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

# 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
```

*formal  
parameter*

*Function  
definition*

```
x = 3
```

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

*actual  
parameter*

*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 )
```

Global scope

# VARIABLE SCOPE

---

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

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

Global scope

f

x

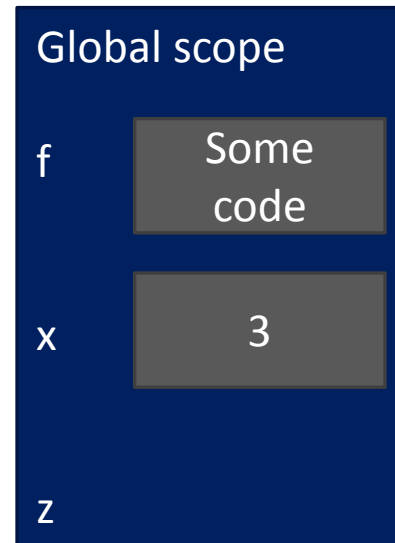
z

# VARIABLE SCOPE

---

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

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

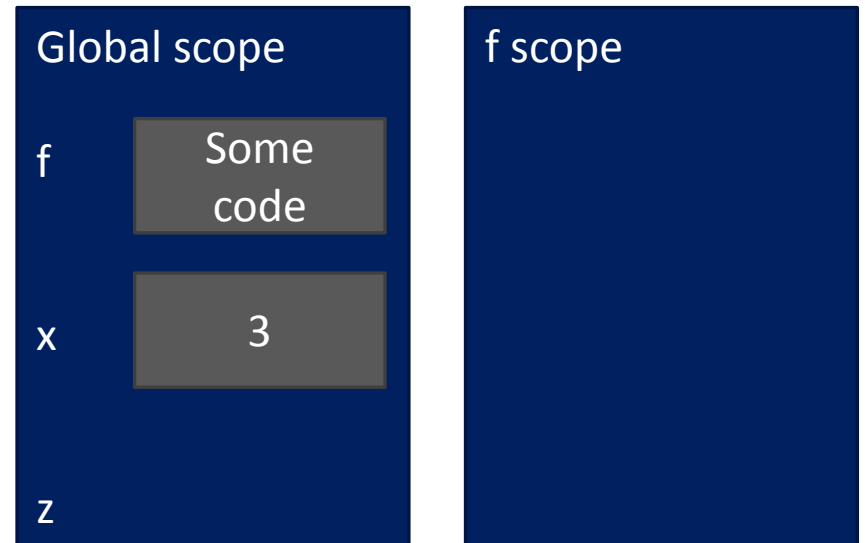


# VARIABLE SCOPE

---

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

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

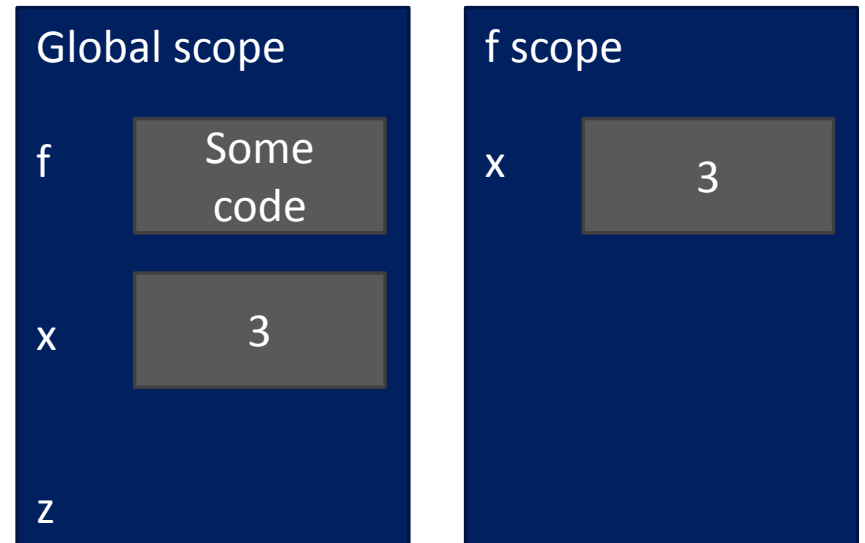


# VARIABLE SCOPE

---

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

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

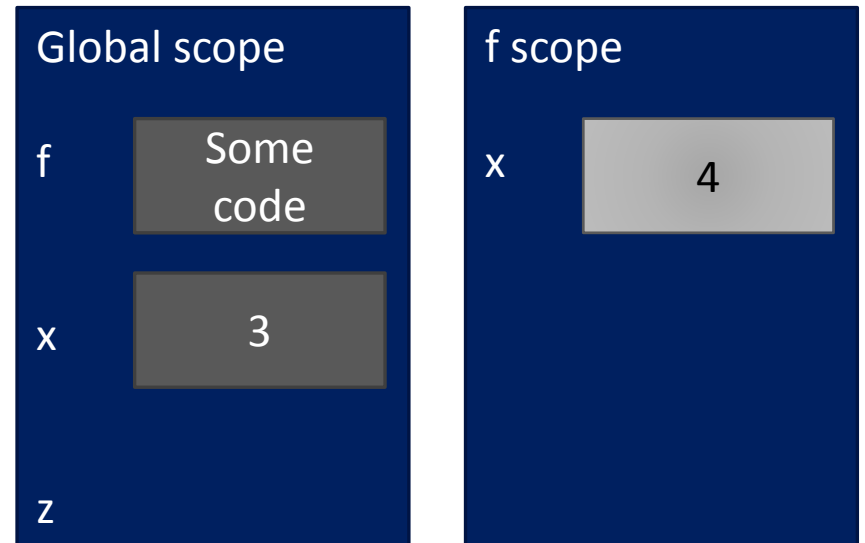


# VARIABLE SCOPE

---

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

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



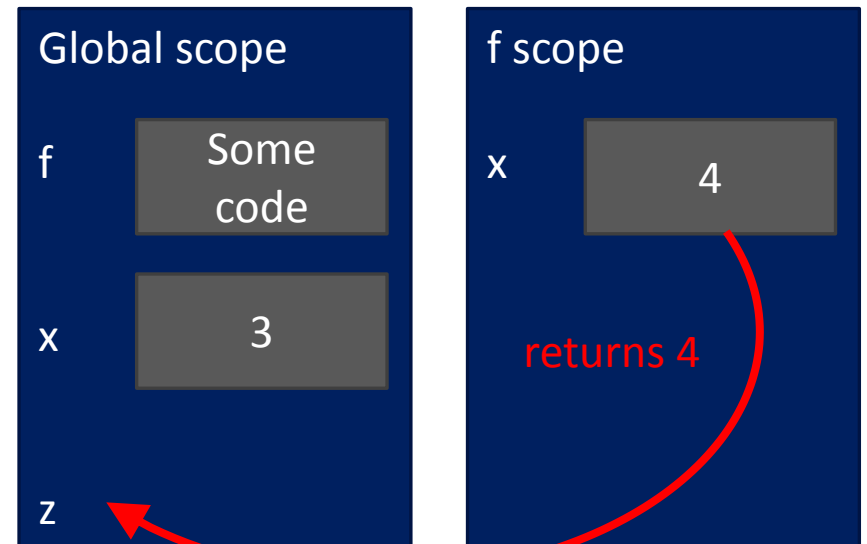


# VARIABLE SCOPE

---

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

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

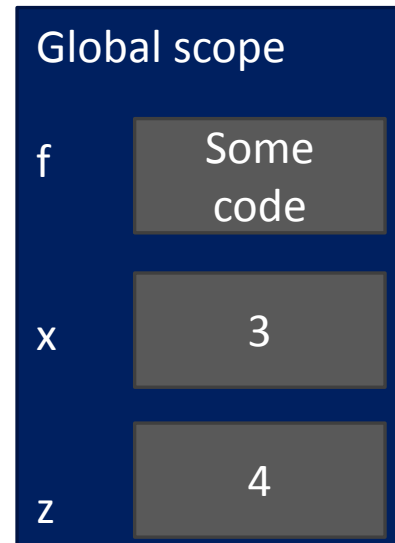


# VARIABLE SCOPE

---

```
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

---

- |   |  |
|---|--|
| <ul style="list-style-type: none"><li>▪ return only has meaning <b>inside</b> a function</li><li>▪ only <b>one</b> return executed inside a function</li><li>▪ code inside function but after return statement not executed</li><li>▪ has a value associated with it, <b>given to function caller</b></li></ul> | <ul style="list-style-type: none"><li>▪ print can be used <b>outside</b> functions</li><li>▪ can execute <b>many</b> print statements inside a function</li><li>▪ code inside function can be executed after a print statement</li><li>▪ has a value associated with it, <b>outputted</b> to the console</li></ul> |
|---|--|

# 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

---

- 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*

# 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*

# 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)
```

Global scope

# 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)
```

Global scope

func\_a

Some  
code

func\_b

Some  
code

func\_c

Some  
code

# 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)
```

Global scope

func\_a

Some  
code

func\_b

Some  
code

func\_c

Some  
code

# 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)
```

Global scope

func\_a

Some  
code

func\_b

Some  
code

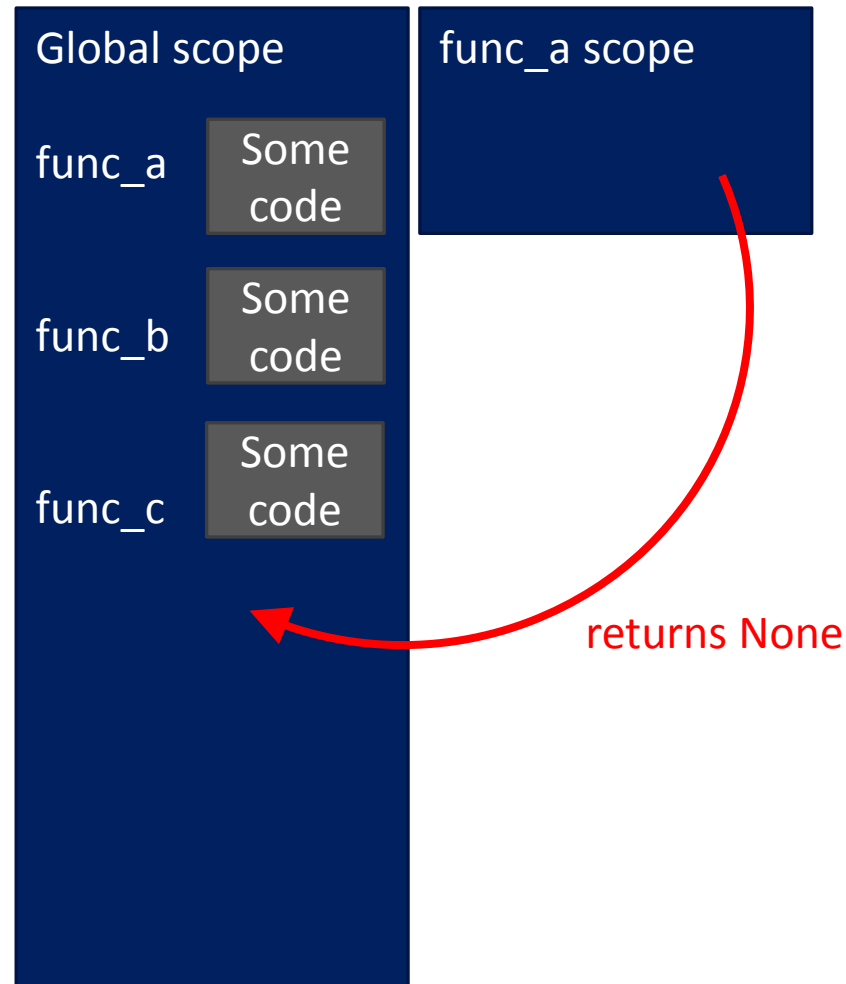
func\_c

Some  
code

func\_a scope

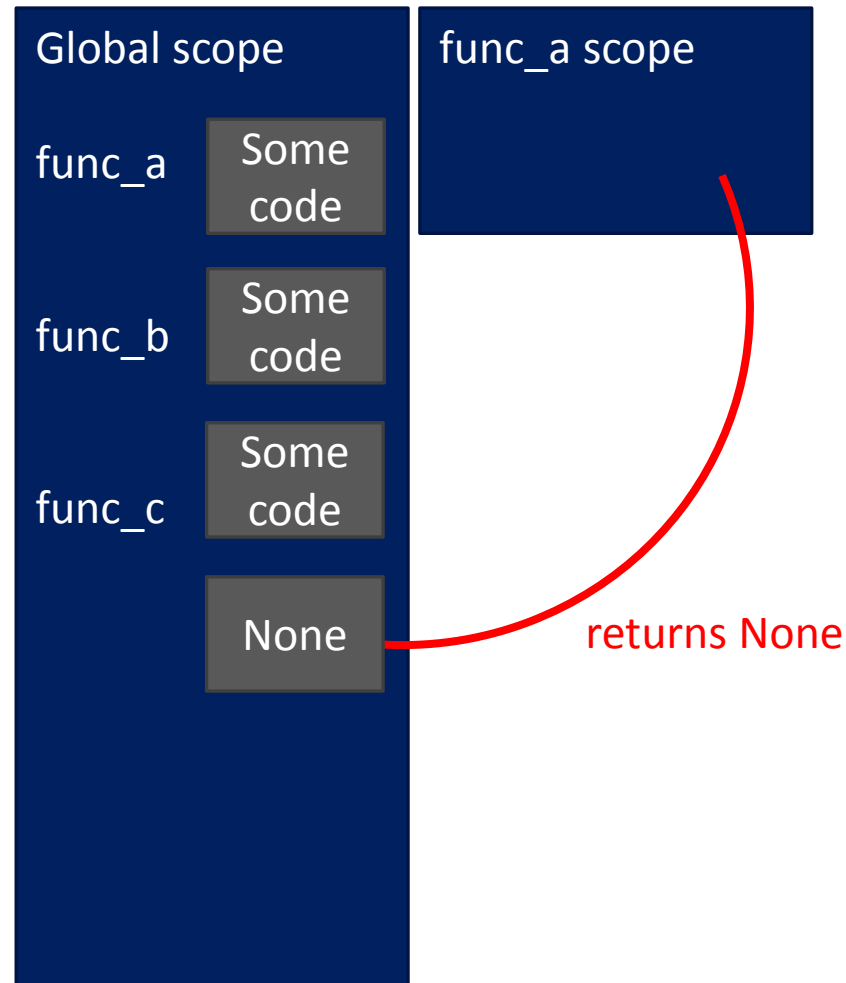
# FUNCTIONS AS ARGUMENTS

```
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)
```

Global scope

func\_a

Some  
code

func\_b

Some  
code

func\_c

Some  
code

None

# 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)
```

Global scope

func\_a

Some  
code

func\_b

Some  
code

func\_c

Some  
code

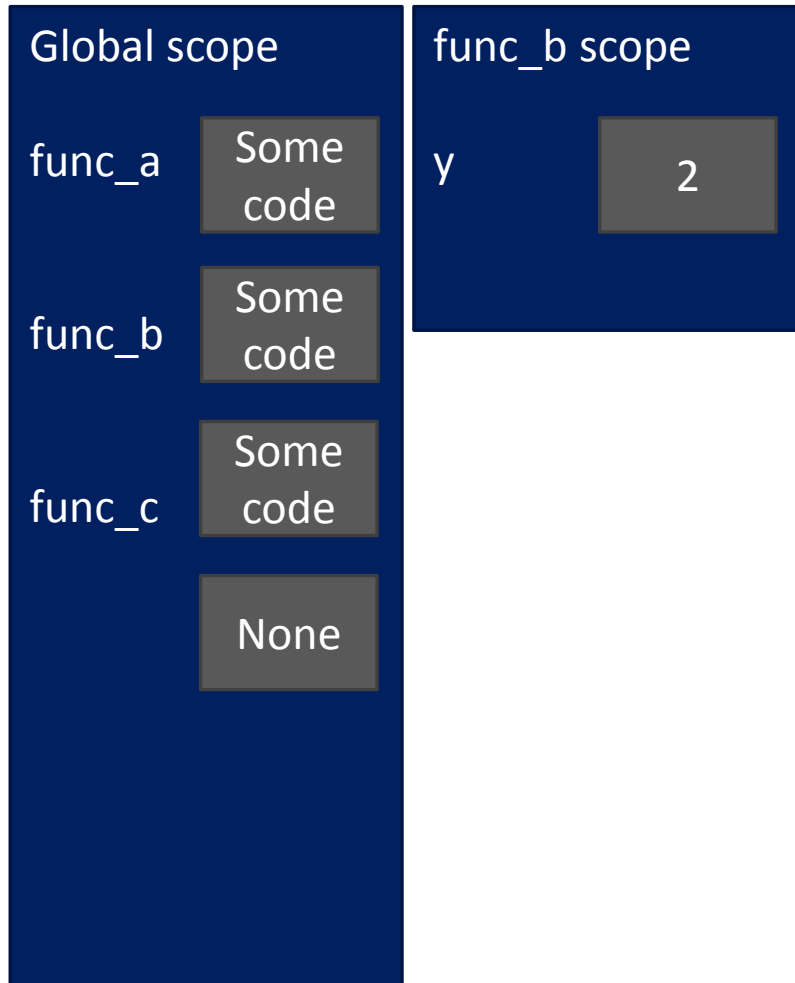
None

func\_b scope



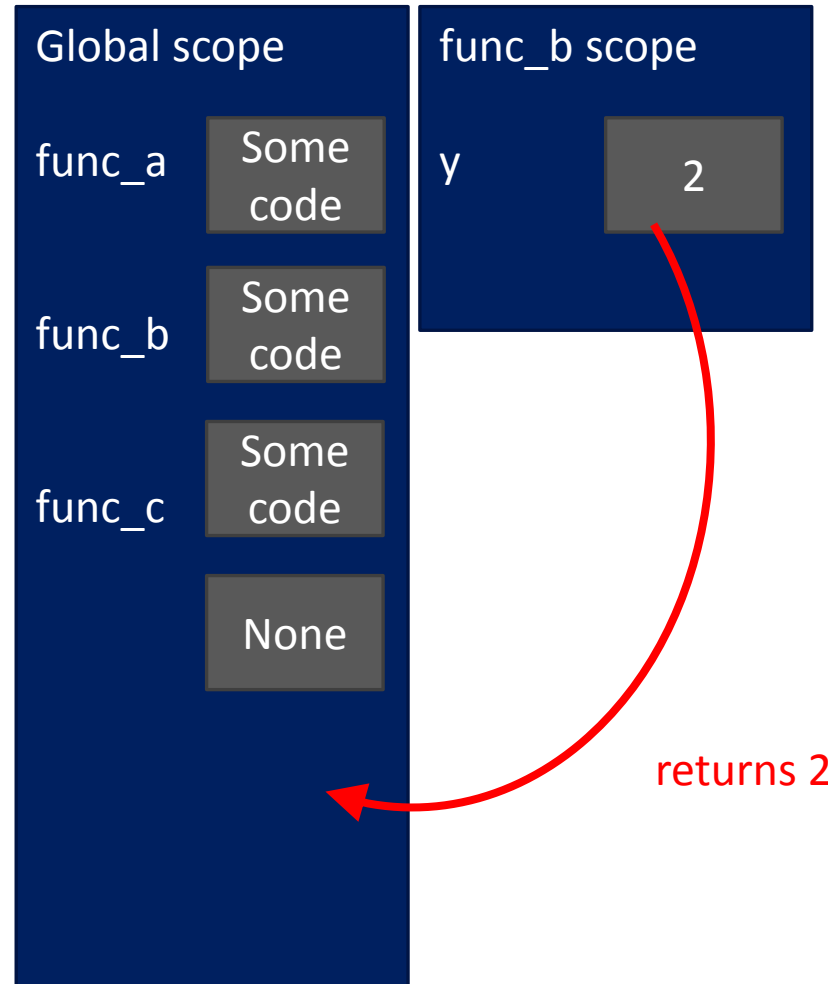
# FUNCTIONS AS ARGUMENTS

```
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)
```



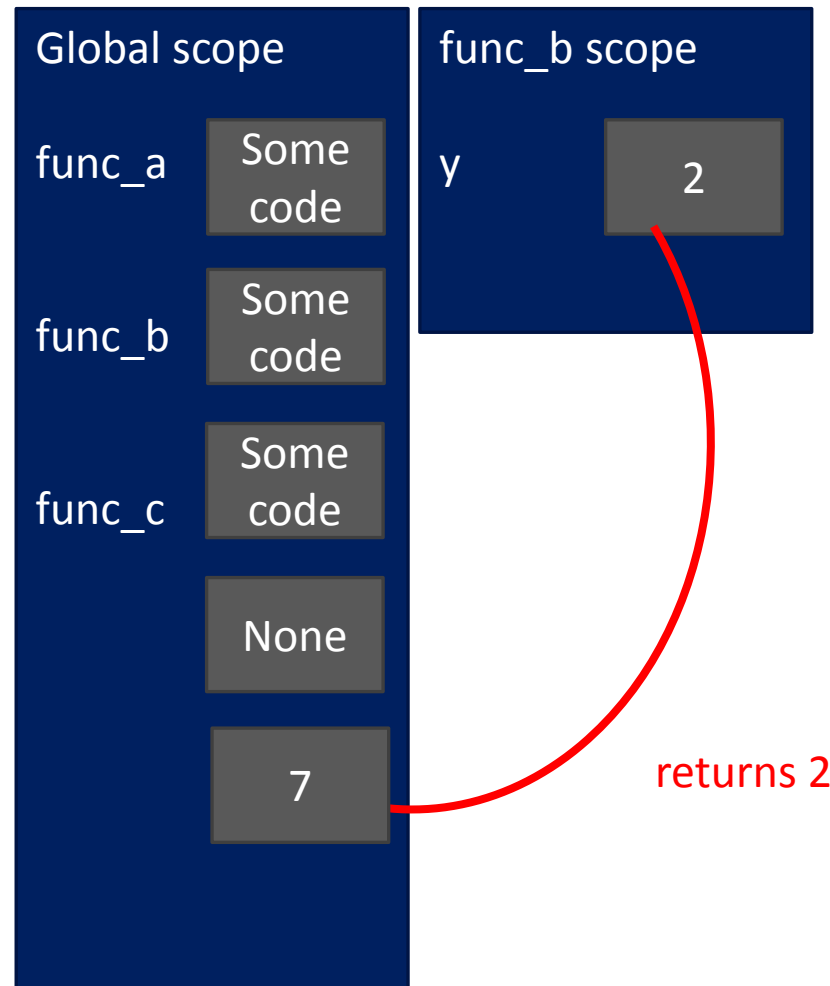
# FUNCTIONS AS ARGUMENTS

```
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)
```



# 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)
```

Global scope

func\_a

Some  
code

func\_b

Some  
code

func\_c

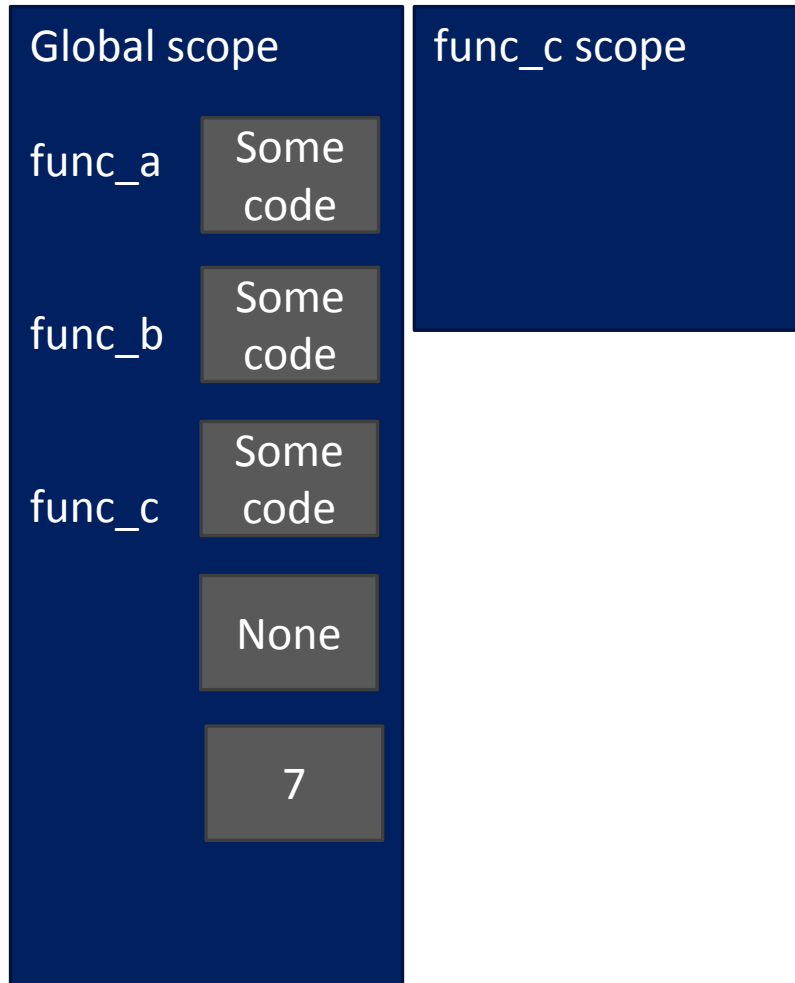
Some  
code

None

7

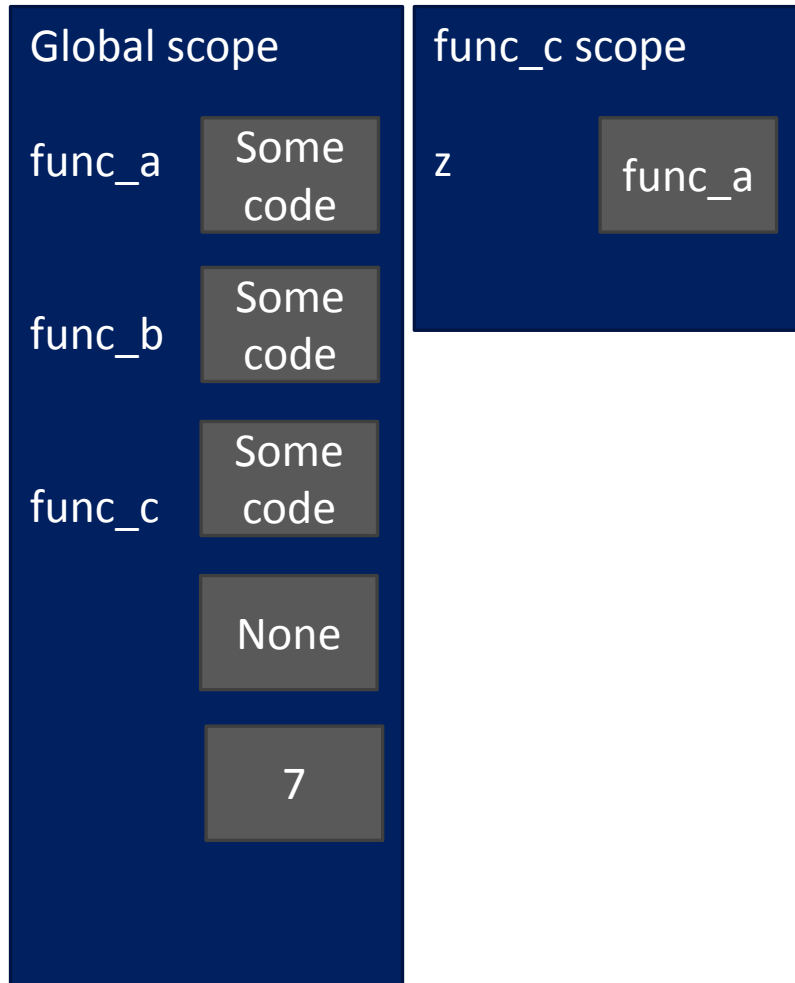
# FUNCTIONS AS ARGUMENTS

```
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)
```



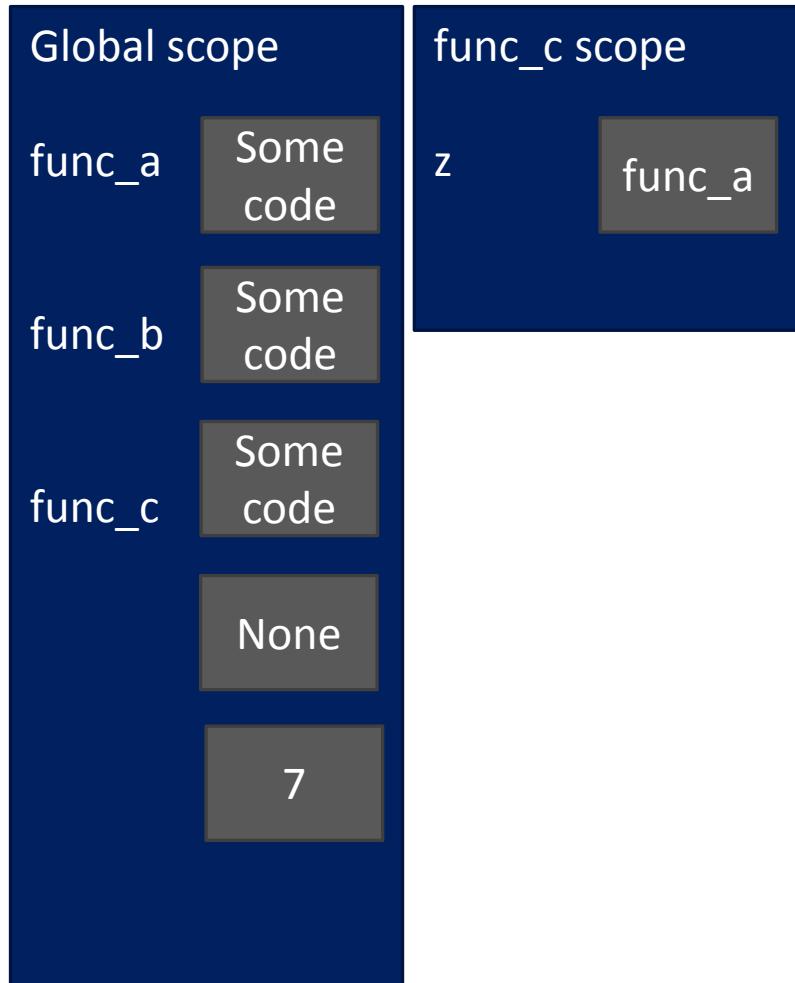
# FUNCTIONS AS ARGUMENTS

```
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)
```



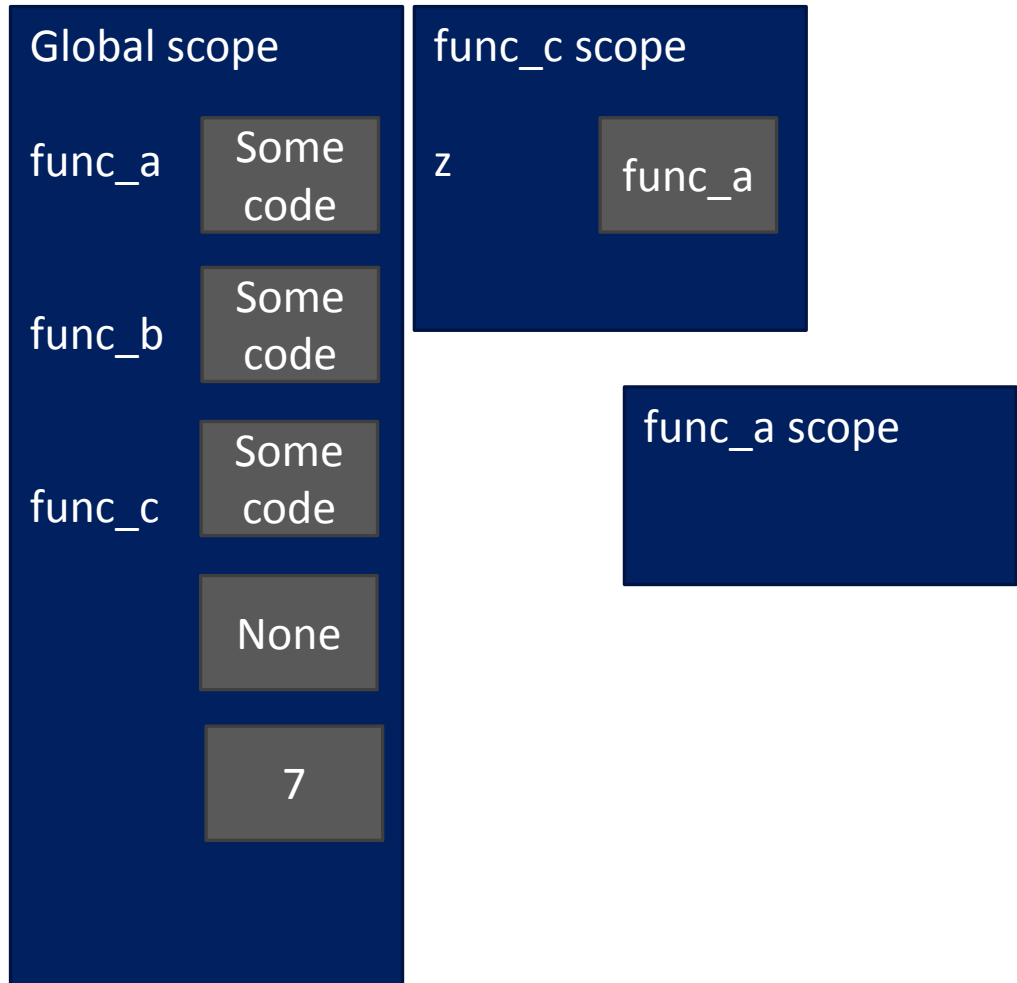
# FUNCTIONS AS ARGUMENTS

```
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)
```



# FUNCTIONS AS ARGUMENTS

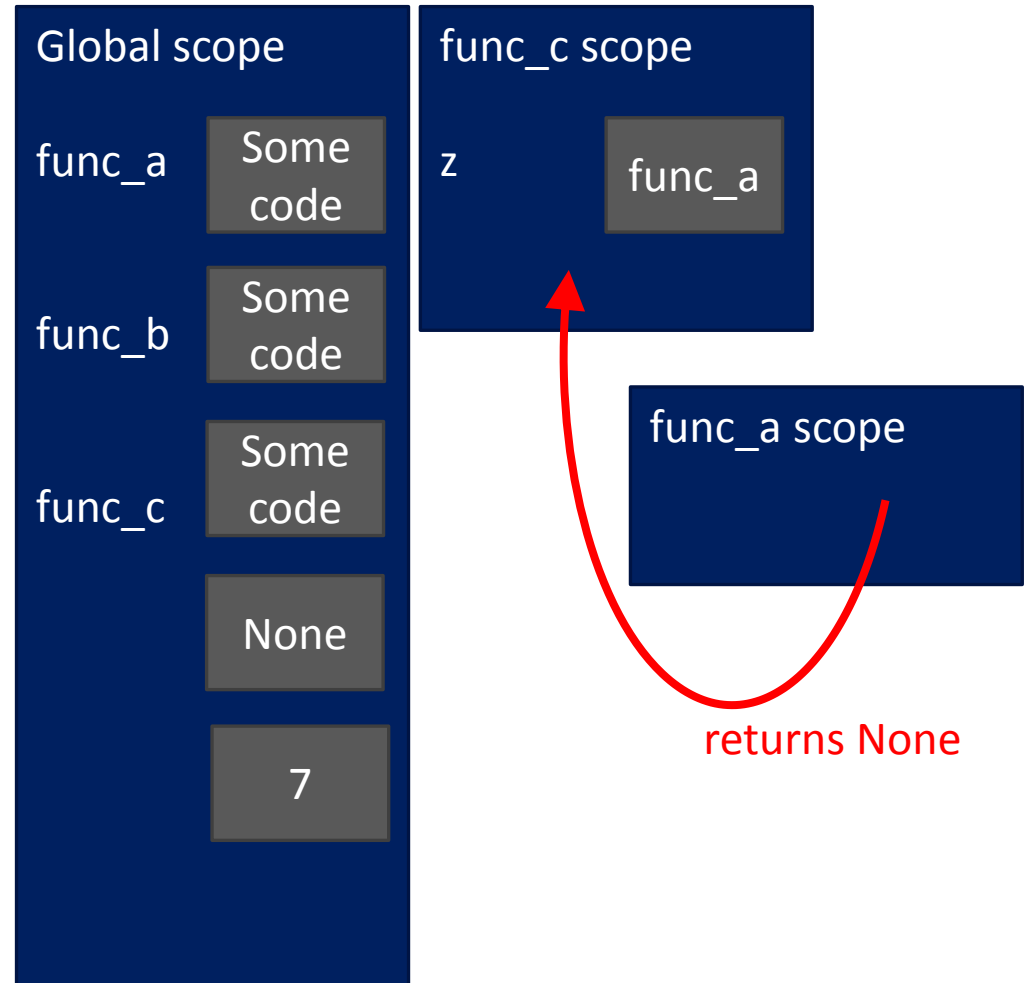
```
def func_a():  
    print 'inside func_a'  
  
def func_b(y):  
    print 'inside func_b'  
    return y  
  
def func_c(z):  
    print 'inside func_c'  
    return z()  
  
print func_a()  
print 5 + func_b(2)  
print func_c(func_a)
```





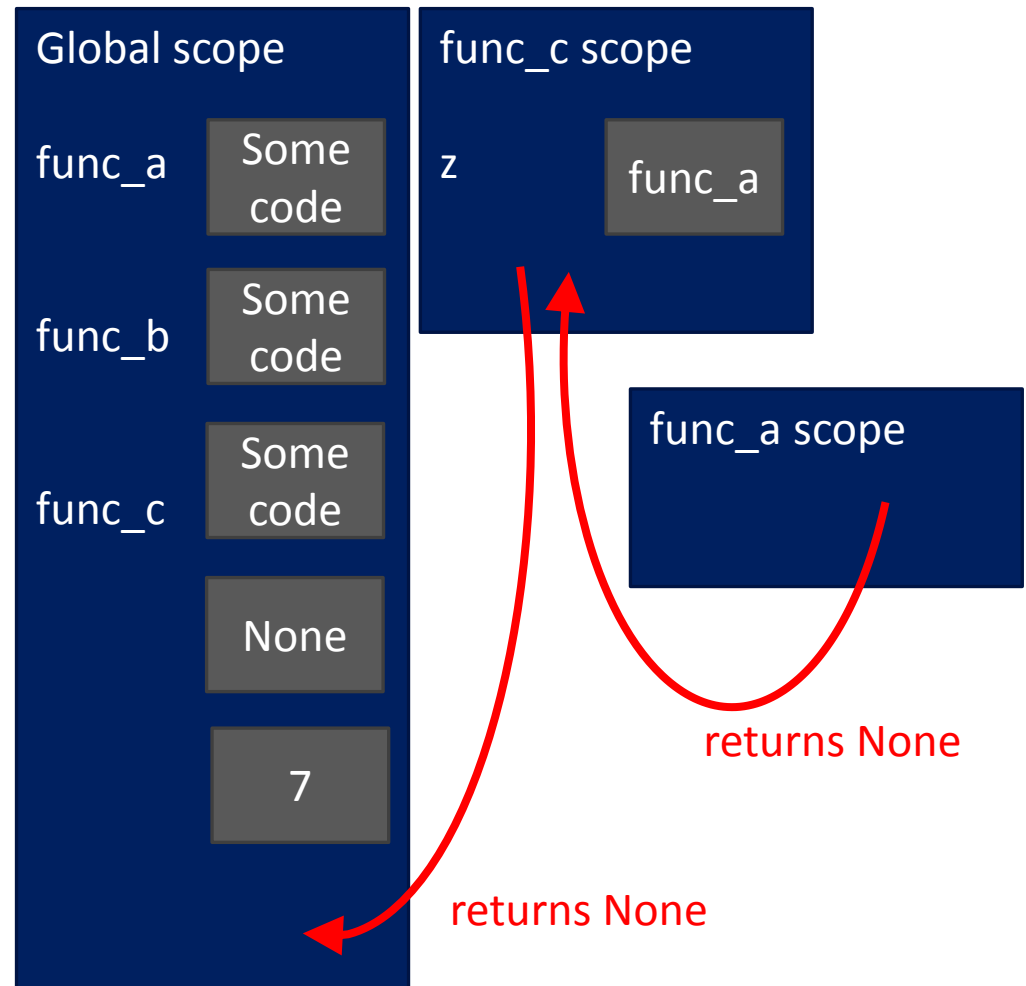
# FUNCTIONS AS ARGUMENTS

```
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)
```



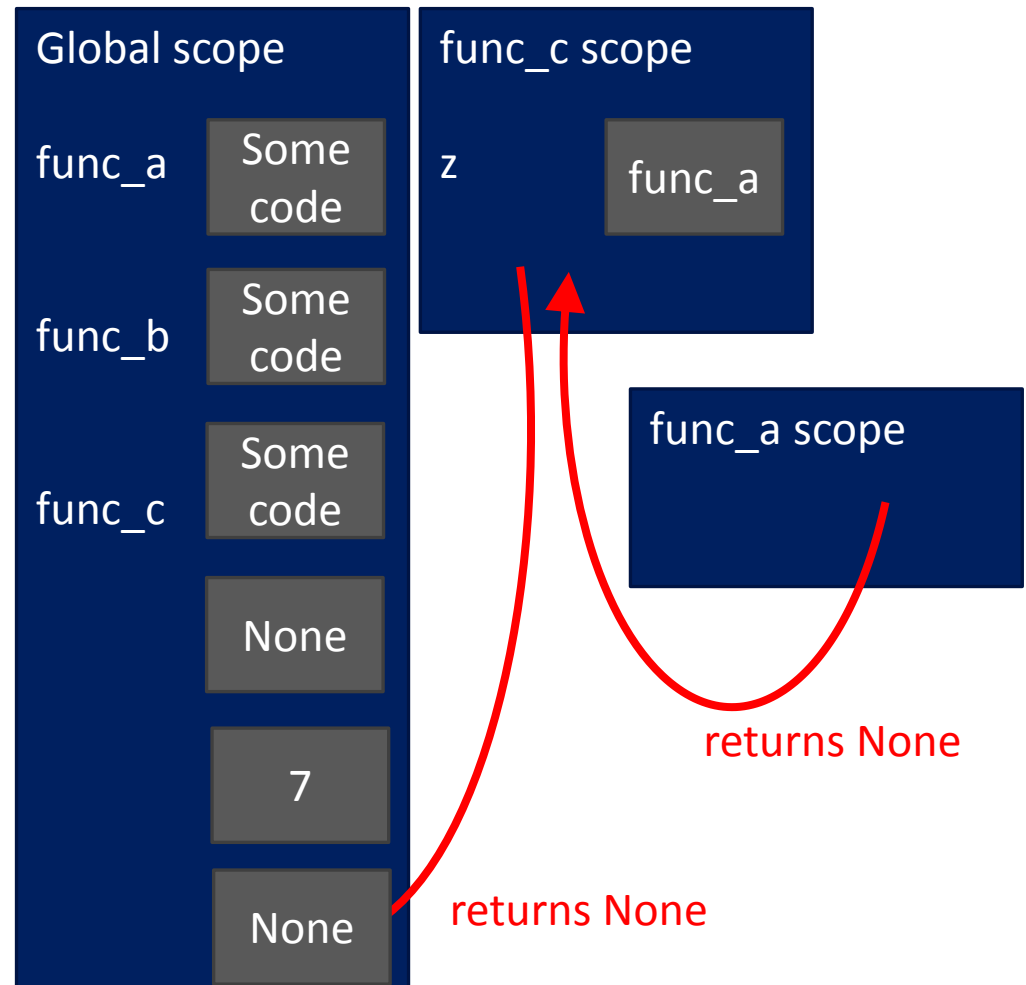
# FUNCTIONS AS ARGUMENTS

```
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)
```



# 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)
```

# 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)
```

# 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 is re-defined  
in scope of f*

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

# 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 is re-defined  
in scope of f*

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

*different x  
objects*

# 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 is re-defined  
in scope of f*

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

*different x  
objects*

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

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



# 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 is re-defined  
in scope of f*

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

*different x  
objects*

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

*x from  
outside g*

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

# 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 is re-defined  
in scope of f*

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

*different x  
objects*

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

*x from  
outside g*

```
x = 5
```

```
g(x)  
print(x)
```

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

# 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 is re-defined  
in scope of f*

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

*different x  
objects*

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

*x from  
outside g*

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

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

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

# 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 is re-defined  
in scope of f*

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

*different x  
objects*

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

*x from  
outside g*

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

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

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

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

# 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 is re-defined  
in scope of f*

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

*different x  
objects*

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

*x from  
outside g*

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

*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



IMPORTANT  
and  
TRICKY!

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

[\*http://www.pythontutor.com/\*](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)
```

Global scope

g

x

z



# SCOPE DETAILS

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

*Some code*

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

Global scope

g

x

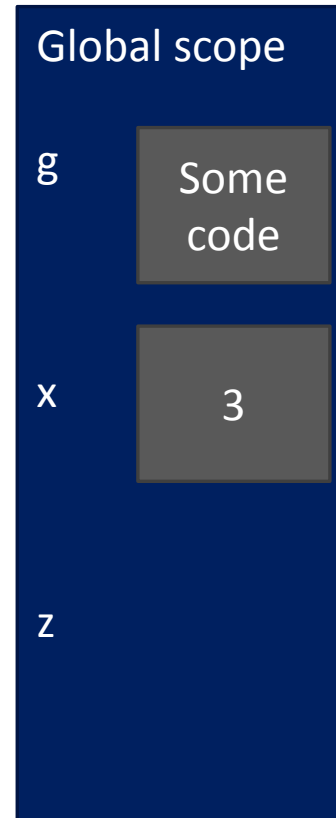
z

# SCOPE DETAILS

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

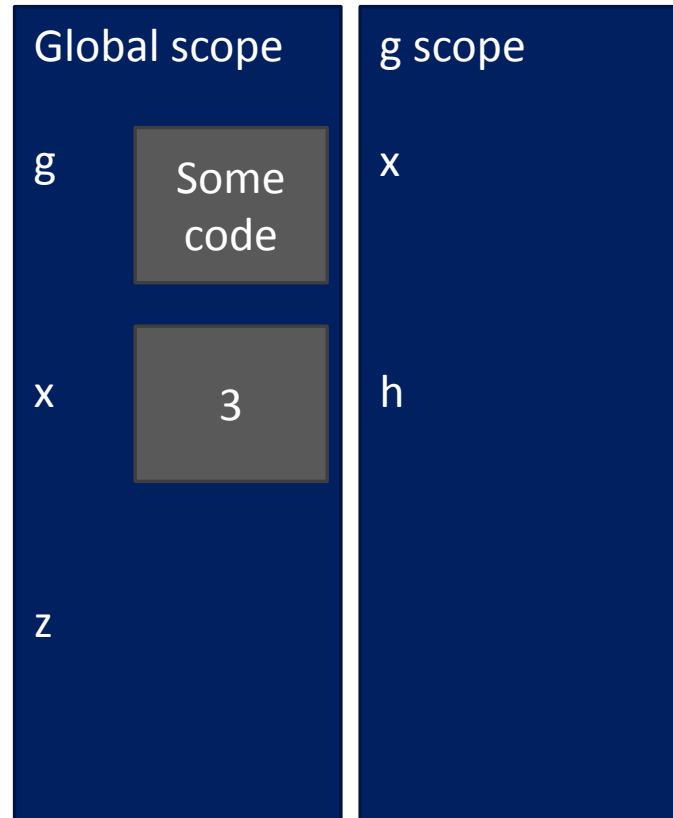
*Some code*

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



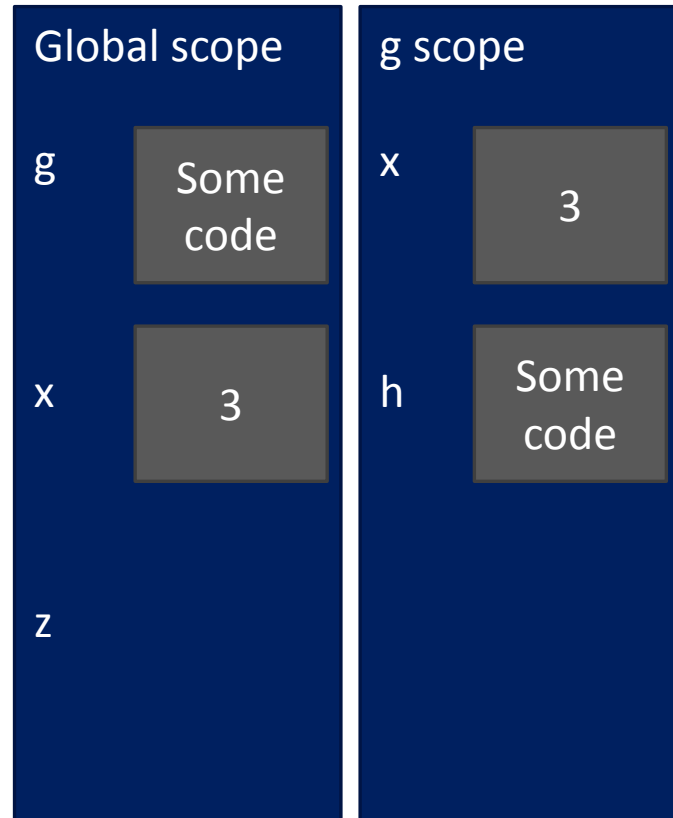
# SCOPE DETAILS

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



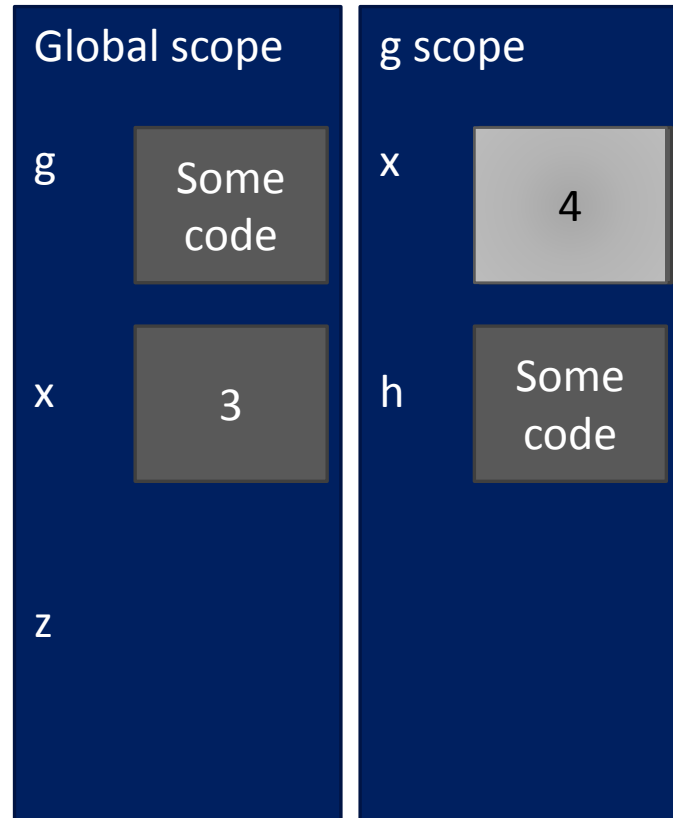
# SCOPE DETAILS

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



# SCOPE DETAILS

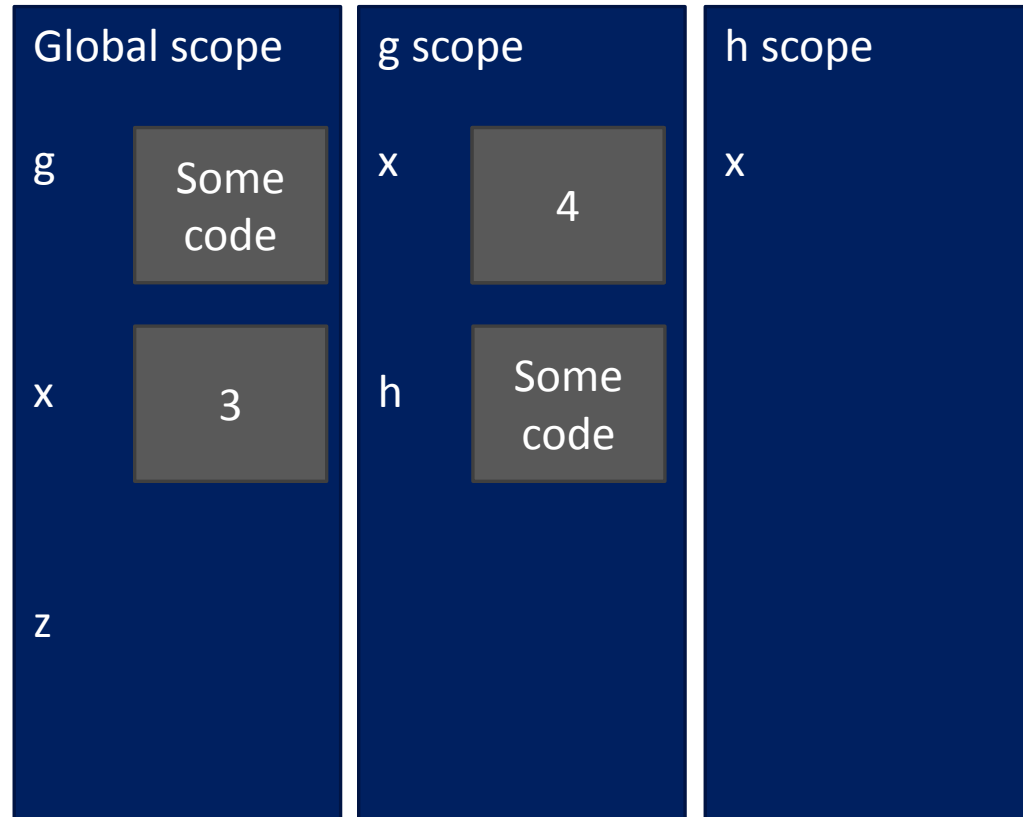
```
def g(x):  
    def h():  
        x = 'abc'  
    x = x + 1  
    print('g: x =', x)  
    h()  
    return x  
  
x = 3  
z = g(x)
```



# SCOPE DETAILS

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

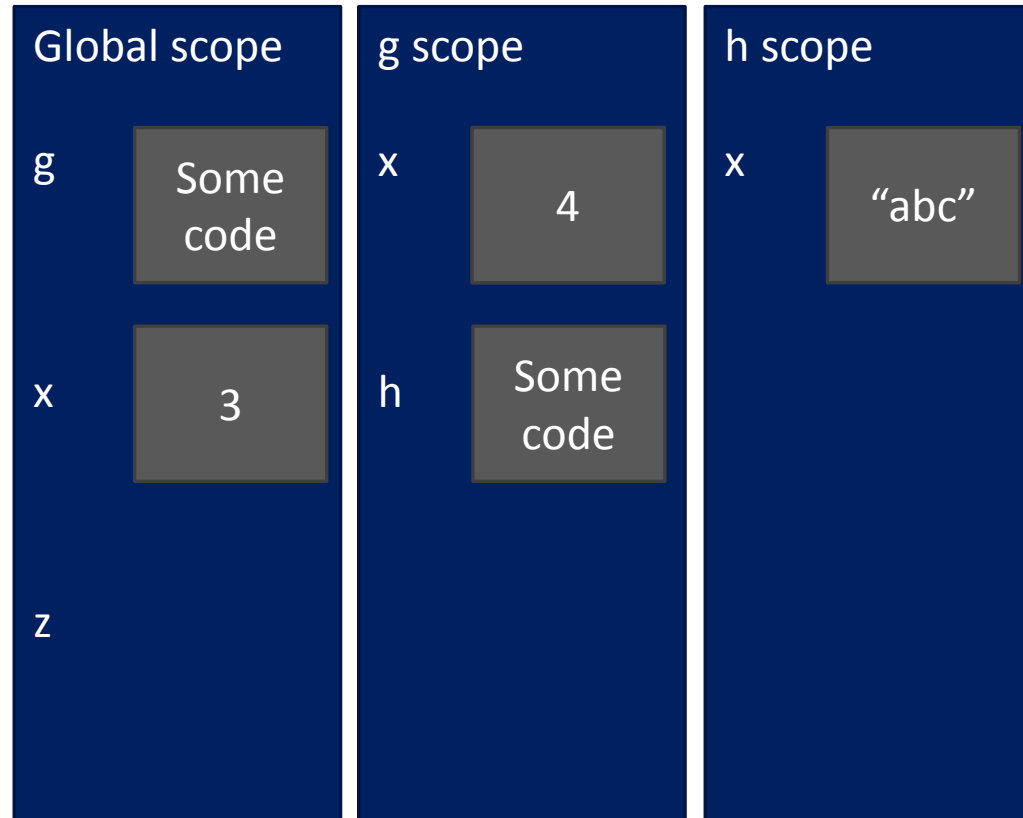
```
x = 3  
z = g(x)
```



# SCOPE DETAILS

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

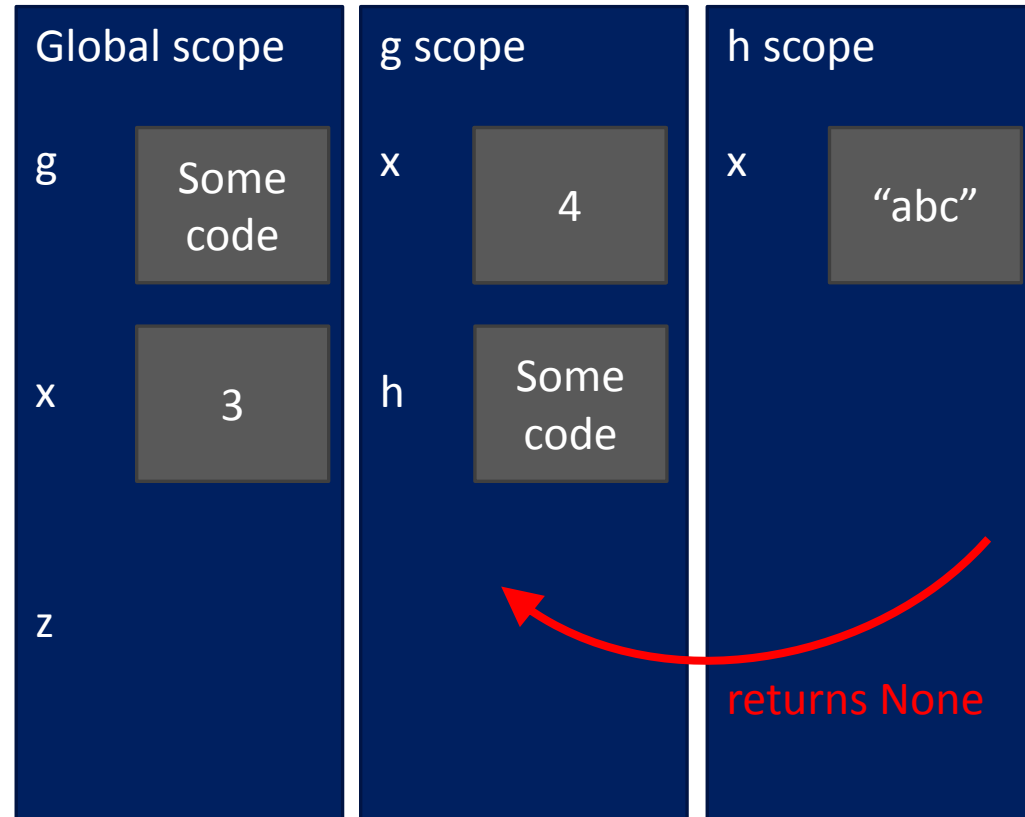
```
x = 3  
z = g(x)
```



# SCOPE DETAILS

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

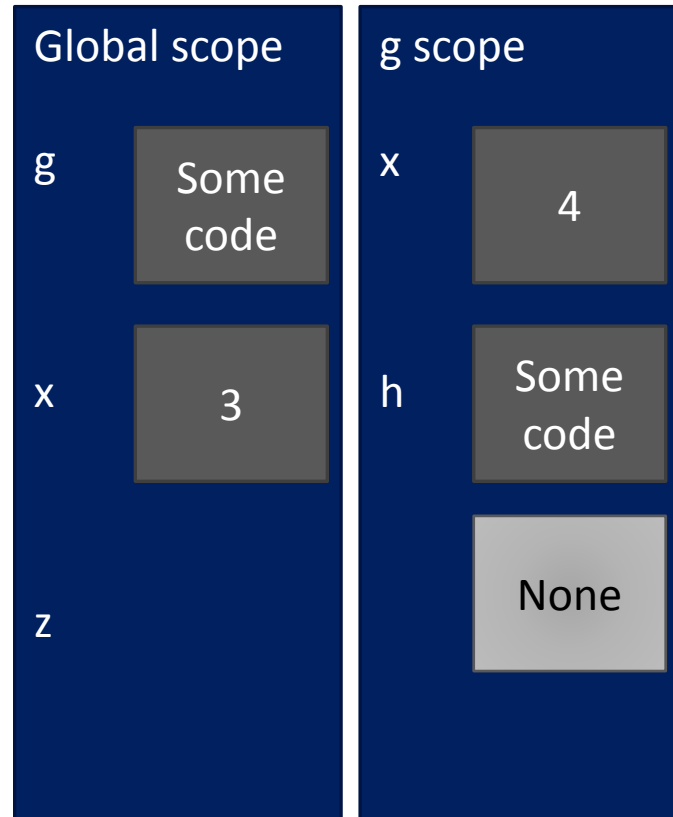
```
x = 3  
z = g(x)
```





# SCOPE DETAILS

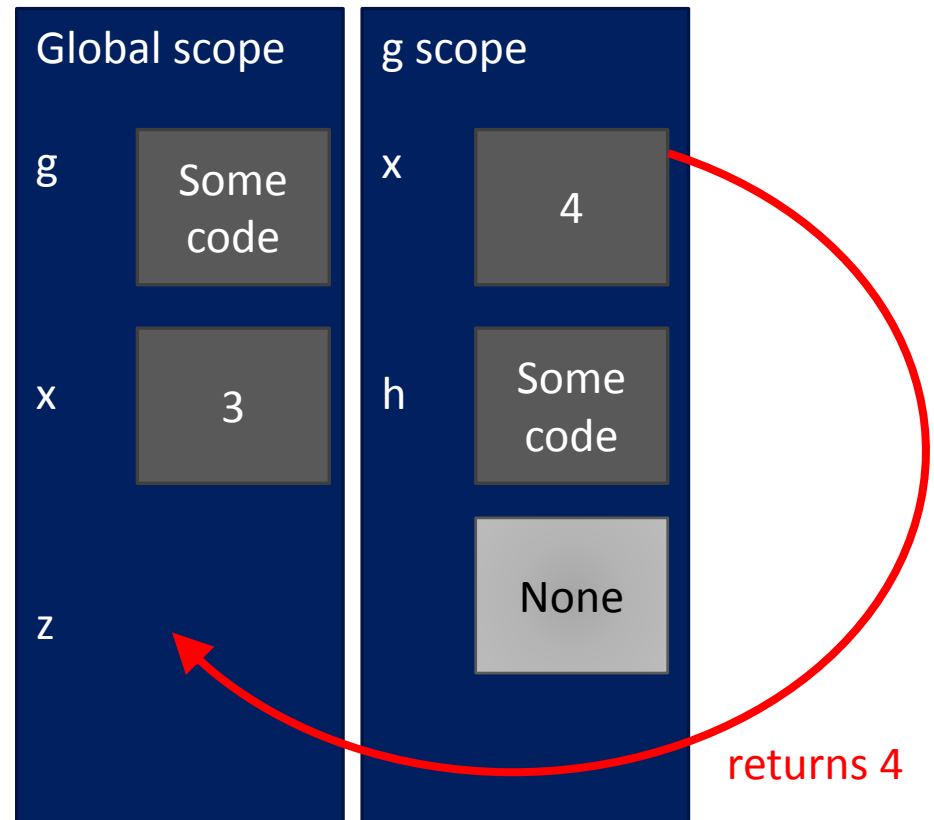
```
def g(x):  
    def h():  
        x = 'abc'  
    x = x + 1  
    print('g: x =', x)  
    h()  
    return x  
  
x = 3  
z = g(x)
```



# SCOPE DETAILS

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

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

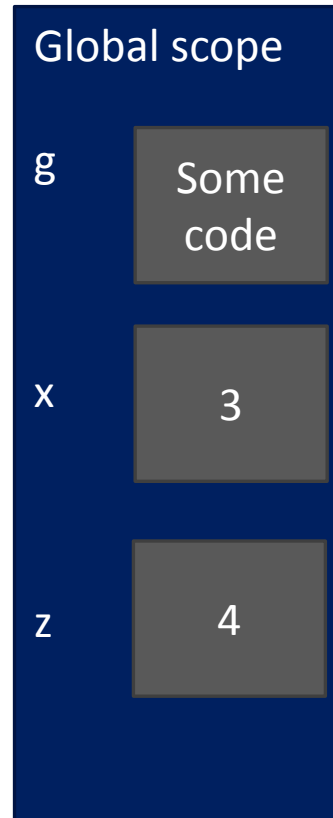


# SCOPE DETAILS

---

```
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!