FUNCTIONS as OBJECTS

(download slides and .py files to follow along)

6.100L Lecture 8

Ana Bell

FUNCTION FROM LAST LECTURE

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

- A function always returns something

WHAT IF THERE IS NO `return` KEYWORD

```
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
  - If invoked in shell, nothing is printed
- No static semantic error generated

```python
def is_even( i ):
    """
    Input: i, a positive int
    Does not return anything
    """
    i%2 == 0
    return None
```

None is a value of type NoneType (not a string, not a number, etc)

A line Python adds implicitly (do not add it yourself)

YOU TRY IT!

- What is printed if you run this code as a file?

```
def add(x,y):
    return x+y
def mult(x,y):
    print(x*y)

add(1,2)
print(add(2,3))
mult(3,4)
print(mult(4,5))
```

`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
- print expression itself returns `None` value

YOU TRY IT!

- Fix the code that tries to write this function

```
def is_triangular(n):
    """ n is an int > 0
    Returns True if n is triangular, i.e. equals a continued
    summation of natural numbers (1+2+3+...+k), False otherwise """
    total = 0
    for i in range(n):
        total += i
        if total == n:
            print(True)
    print(False)
```

FUNCTIONS SUPPORT MODULARITY

- Here is our bisection square root method as a function

```
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    ans = (high + low)/2.0
    while abs(ans**2 - x) >= epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2.0
    # print(ans, 'is close to the root of', x)
    return ans
```

Initialize variables

guess not close enough

iterate

update low or high, depends on guess too small or too large

new value for guess

return result

FUNCTIONS SUPPORT MODULARITY

- Call it with different values

```
print(bisection_root(4))
print(bisection_root(123))
```

- Write a function that calls this one!

YOU TRY IT!

- Write a function that satisfies the following specs

```
def count_nums_with_sqrt_close_to (n, epsilon):
    """ n is an int > 2
        epsilon is a positive number < 1
    Returns how many integers have a square root within epsilon of n """
```

Use `bisection_root` we already wrote to get an approximation for the sqrt of an integer.

For example: `print(count_nums_with_sqrt_close_to(10, 0.1))`

prints 4 because all these integers have a sqrt within 0.1

  - sqrt of `99` is `9.949699401855469`
  - sqrt of `100` is `9.999847412109375`
  - sqrt of `101` is `10.049758911132812`
  - sqrt of `102` is `10.09945678710937 5`

ZOOMING OUT

This is my "black box"

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        if i%2 == 1:
            sum_of_odds += i
    return sum_of_odds

low = 2
high = 7
my_sum = sum_odd(low, high)
```

One function call

Program Scope

sum_odd	function object
low	2
high	7
my_sum	

ZOOMING OUT

a = 2 b = 7

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        if i%2 == 1:
            sum_of_odds += i
    return sum_of_odds

low = 2
high = 7
my_sum = sum_odd(low, high)
```

Program Scope

sum_odd	function object
low	2
high	7
my_sum	

ZOOMING OUT

This is my "black box"

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        if i%2 == 1:
            sum_of_odds += i
    return sum_of_odds

low = 2
high = 7
my_sum = sum_odd(low, high)
```

15

Program Scope	
sum_odd	function object
low	2
high	7
my_sum	15

FUNCTION SCOPE

UNDERSTANDING FUNCTION CALLS

- How does Python execute a function call?
- How does Python know what value is associated with a variable name?
- It **creates a new environment** with every function call!
  - Like a **mini program** that it needs to complete
  - The mini program runs with **assigning its parameters** to some inputs
  - It does the work (aka the **body** of the function)
  - It **returns** a value
  - The **environment disappears** after it returns the value

ENVIRONMENTS

- **Global** environment
  - Where user interacts with Python interpreter
  - Where the program starts out
- Invoking a function creates a **new environment** (frame/scope)

VARIABLE SCOPE

- **Formal parameters** get bound to the value of **input parameters**
- **Scope** is a mapping of names to objects
  - Defines context in which body is evaluated
  - Values of variables given by bindings of names
- Expressions in body of function evaluated wrt this new scope

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

y = 3
z = f( y )
```

formal parameter

Function definition

actual parameter

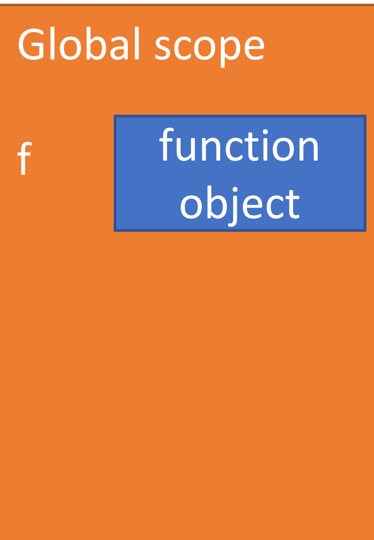
Main program code
* initializes a variable x
* makes a function call f(x)
* assigns return of function to variable z

Can be any legal value

VARIABLE SCOPE after evaluating def

This is my "black box"

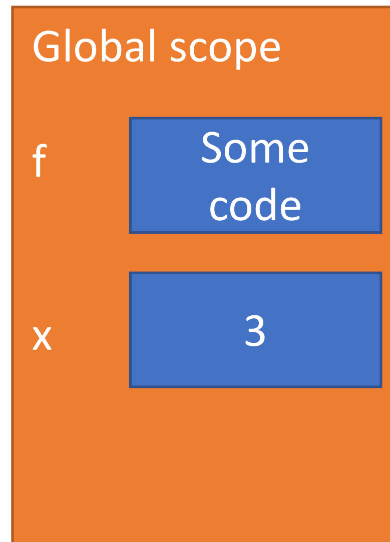
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

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



VARIABLE SCOPE after exec 1st assignment

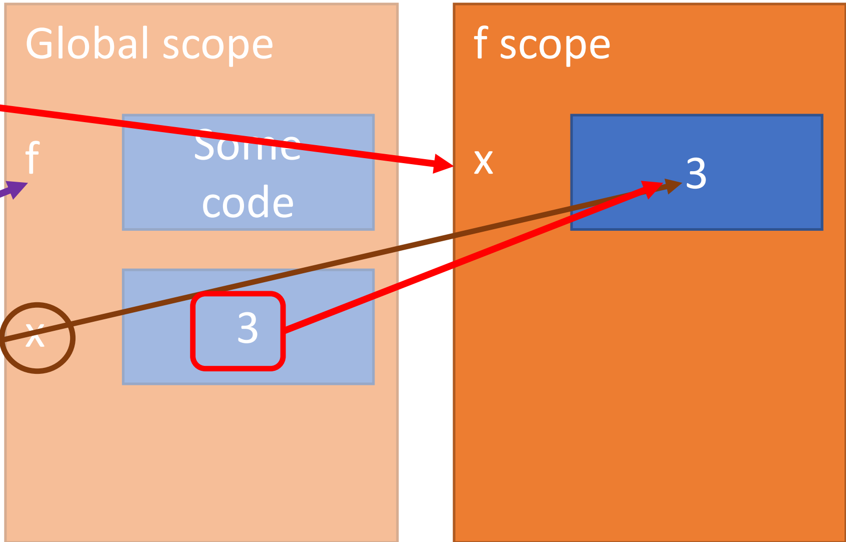
This is my “black box”

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

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



VARIABLE SCOPE
after f invoked

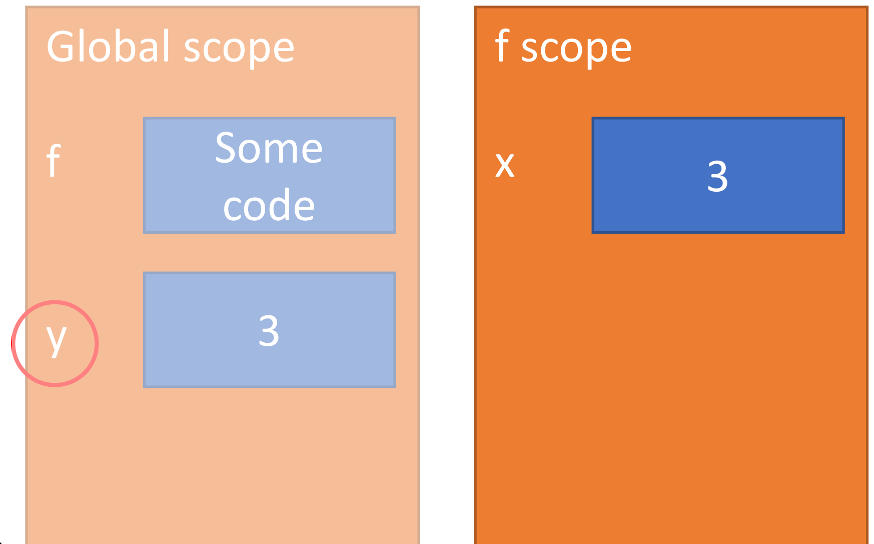
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

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



VARIABLE SCOPE
after f invoked

Name of variable irrelevant, only value important. You can also pass in the value directly.

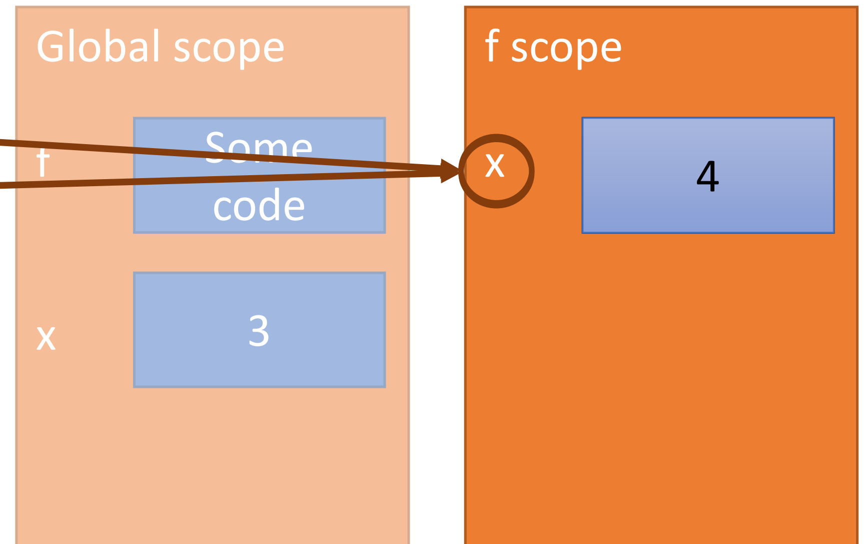
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

y = 3
z = f( y )
```



VARIABLE SCOPE
eval body of f in f's scope

in f(x): x = 4 **printed out**

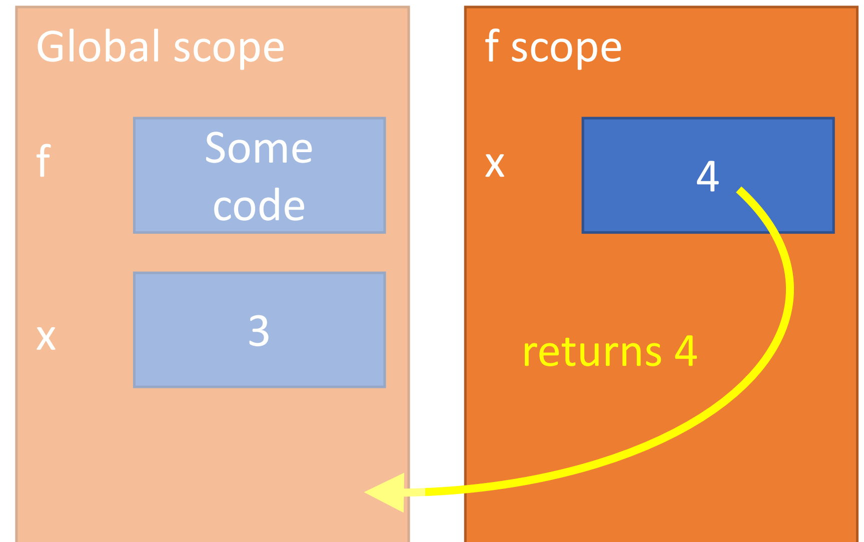
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

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



VARIABLE SCOPE during return

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

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

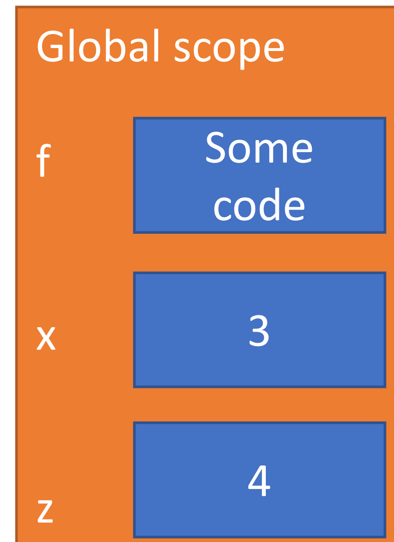
Function call replaced with return value



VARIABLE SCOPE after exec 2nd assignment

If I now ask for value of x in Python interpreter, it will print 3

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

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



BIG IDEA

You need to know what expression you are executing to know the scope you are in.

ANOTHER SCOPE EXAMPLE

- Inside a function, **can access** a variable defined outside
- Inside a function, **cannot modify** a variable defined outside (can by using **global variables**, but frowned upon)
- Use the Python Tutor to step through these!

```
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

Output:
2
5

```
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

Output:
5
6
5

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

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

UnboundLocalError: local variable 'x' referenced before assignment

Error

FUNCTIONS as ARGUMENTS

HIGHER ORDER PROCEDURES

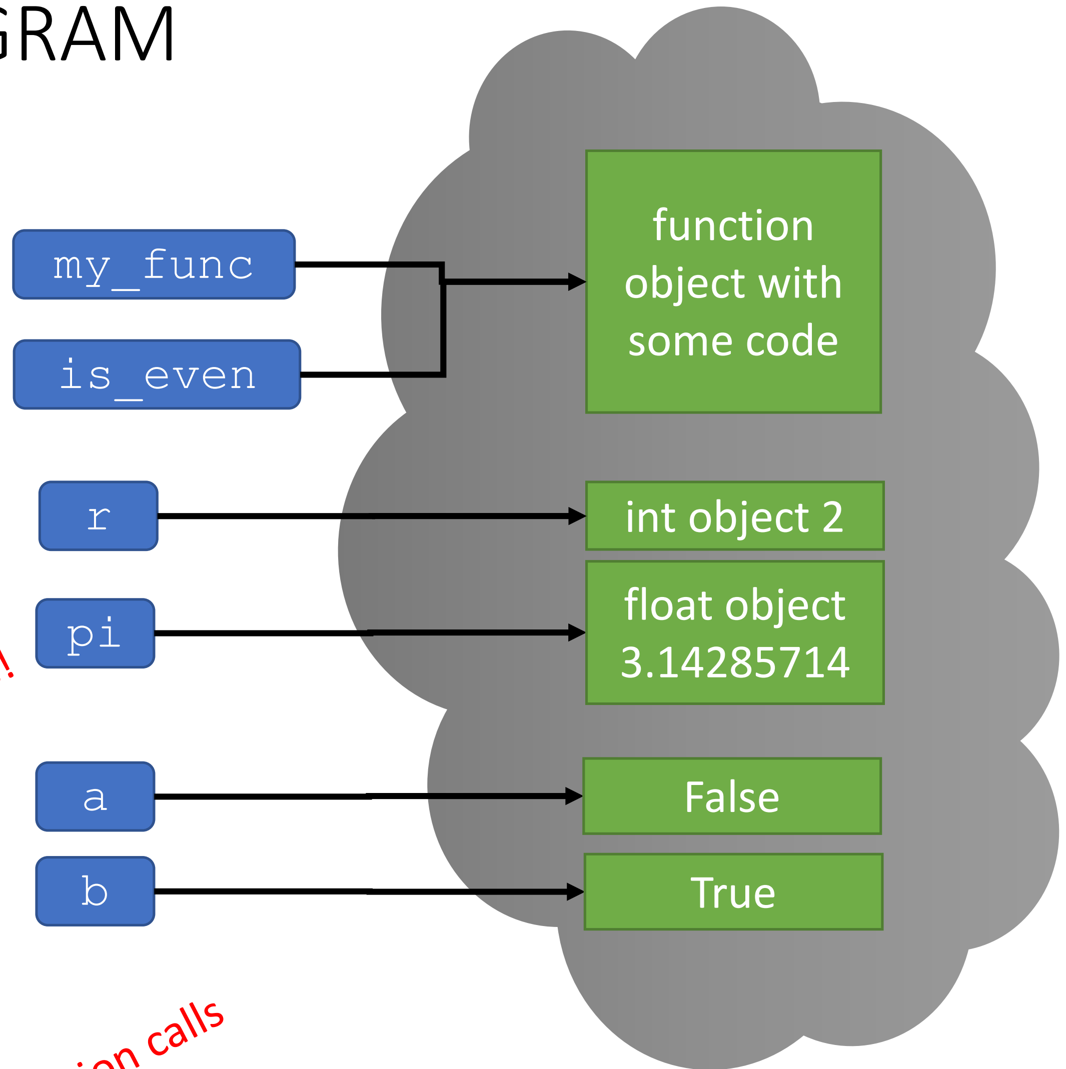
- Objects in Python have a type
  - int, float, str, Boolean, NoneType, function
- Objects can appear in RHS of assignment statement
  - Bind a name to an object
- Objects
  - Can be **used as an argument** to a procedure
  - Can be **returned as a value** from a procedure
- Functions are also **first class objects!**
- Treat functions just like the other types
  - Functions can be arguments to another function
  - Functions can be returned by another function

OBJECTS IN A PROGRAM

```
def is_even(i):
    return i%2 == 0

r = 2

pi = 22/7

my_func = is_even

a = is_even(3)

b = my_func(4)
```

NOT a function call, just names!

Two function calls



BIG IDEA

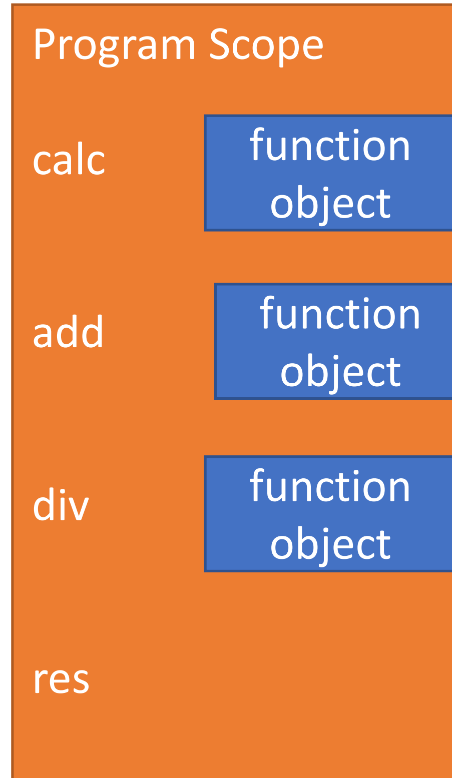
Everything in Python is an object.

FUNCTION AS A PARAMETER

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denominator was 0.")

print(calc(add, 2, 3))
```

STEP THROUGH THE CODE

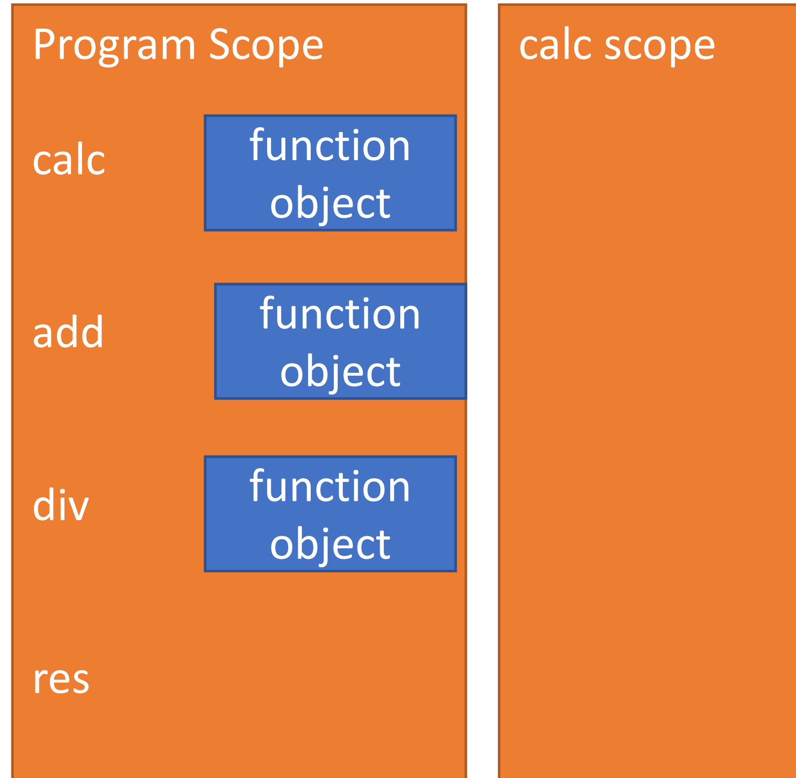
```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



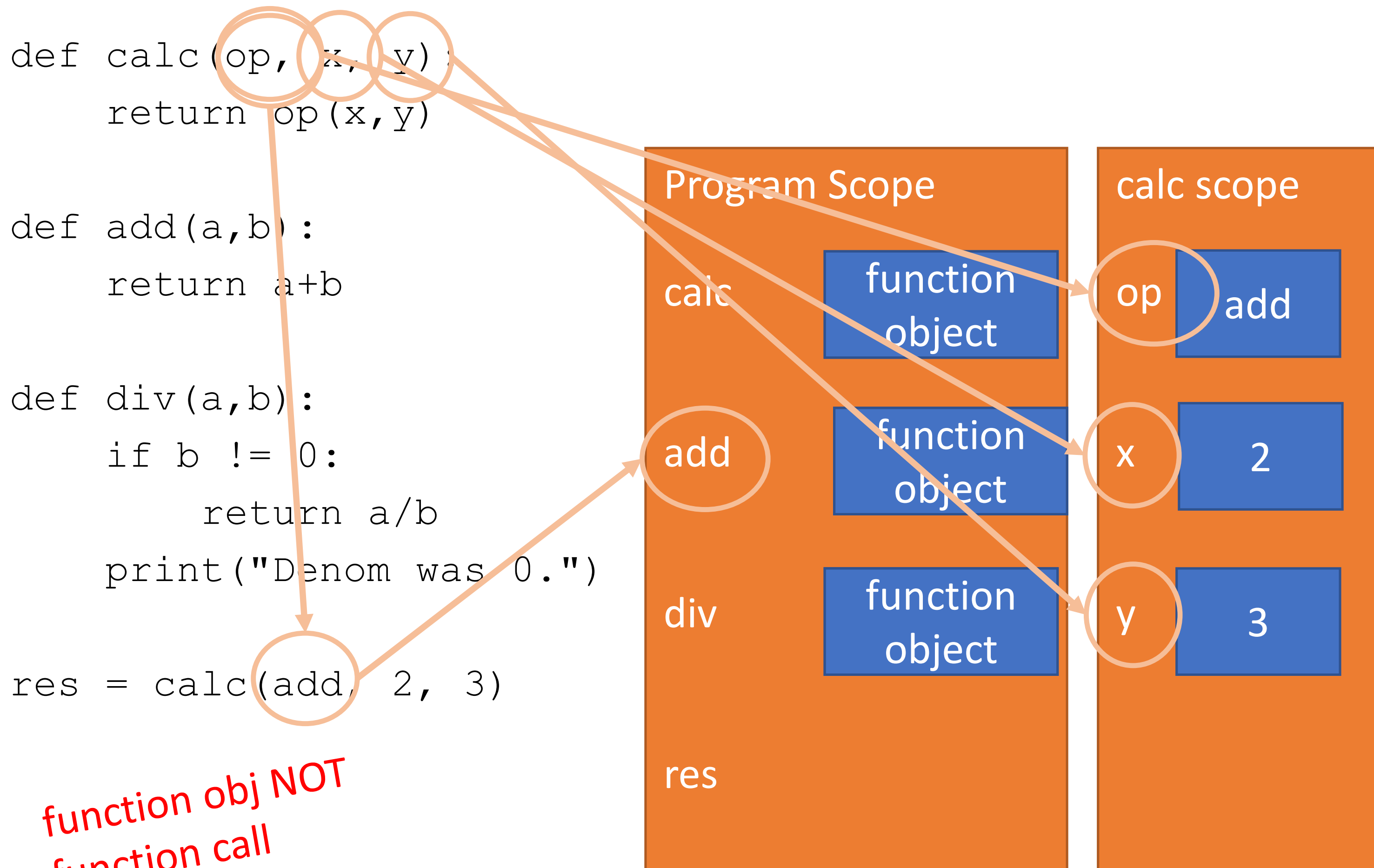
CREATE calc SCOPE

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```

Function call



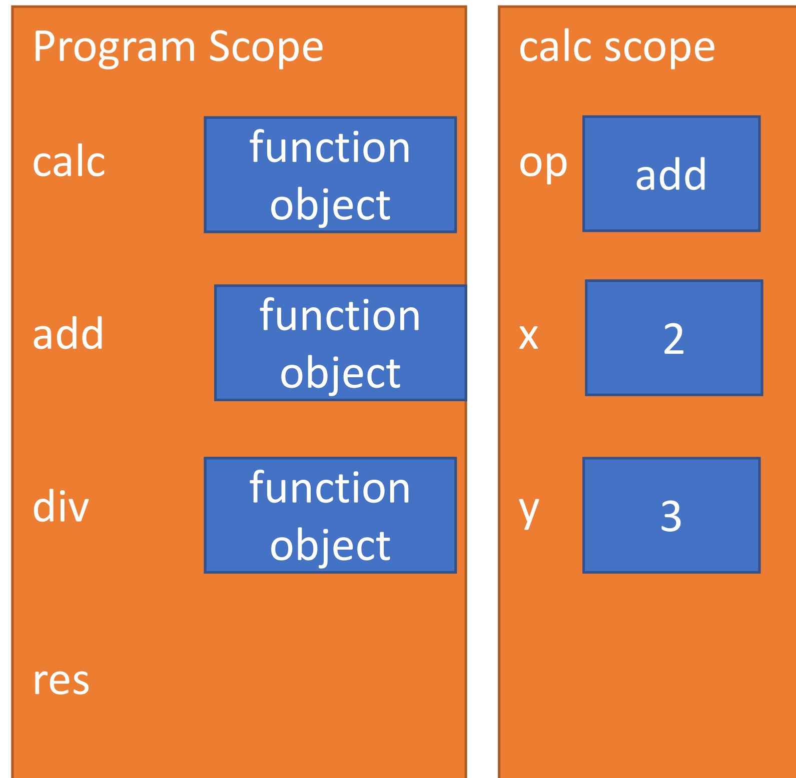
MATCH FORMAL PARAMS in calc



FIRST (and only) LINE IN calc

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```

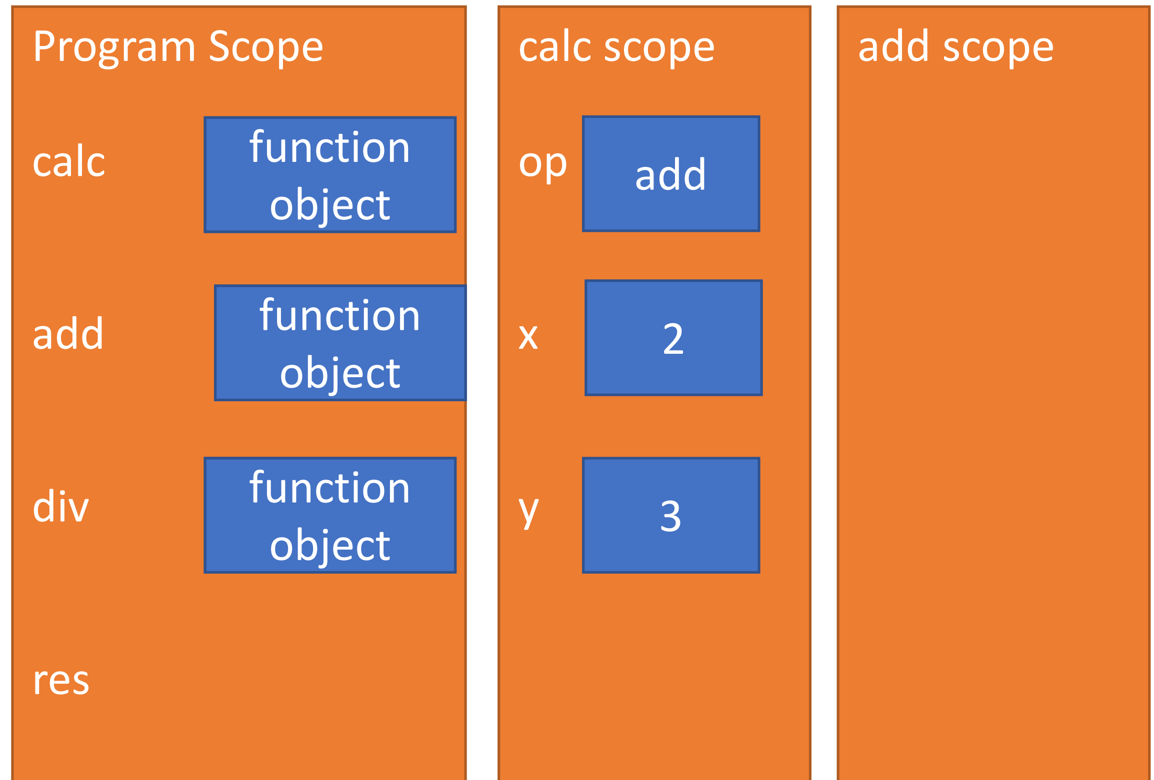
return add(2,3)
Just replace each param with its value



CREATE SCOPE OF add

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```

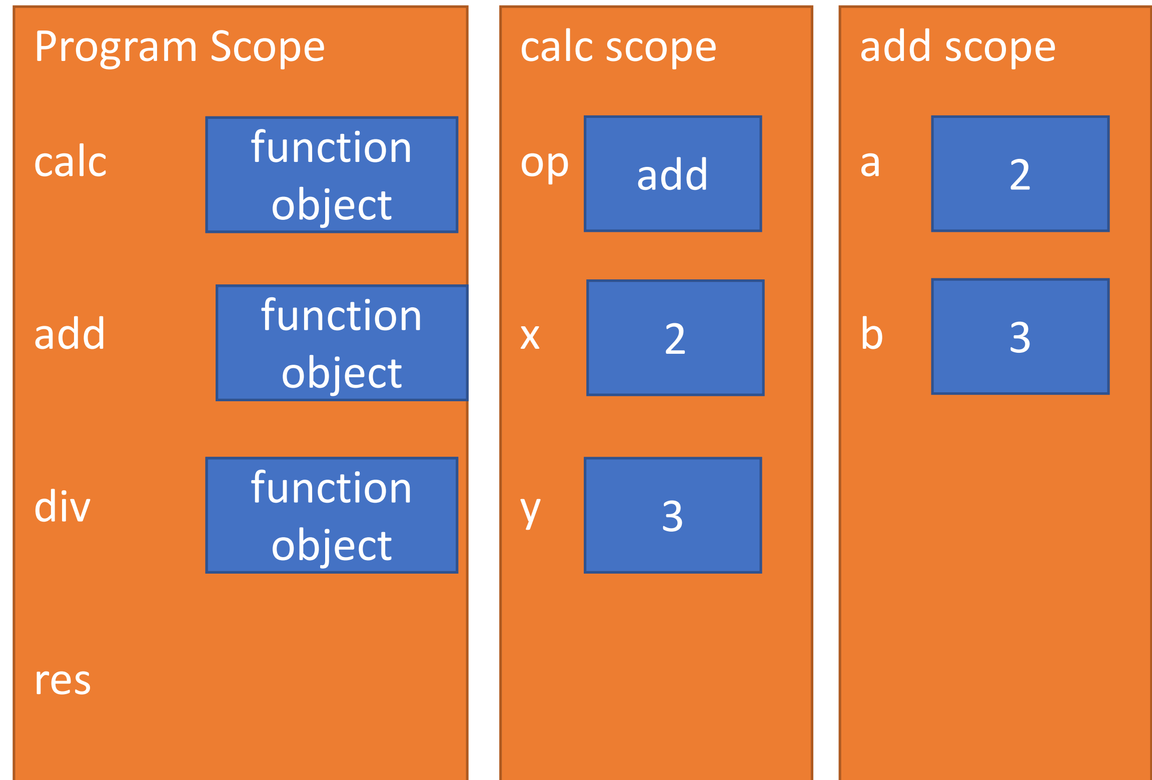
Function call in calc scope: add(2,3)



MATCH FORMAL PARAMS IN add

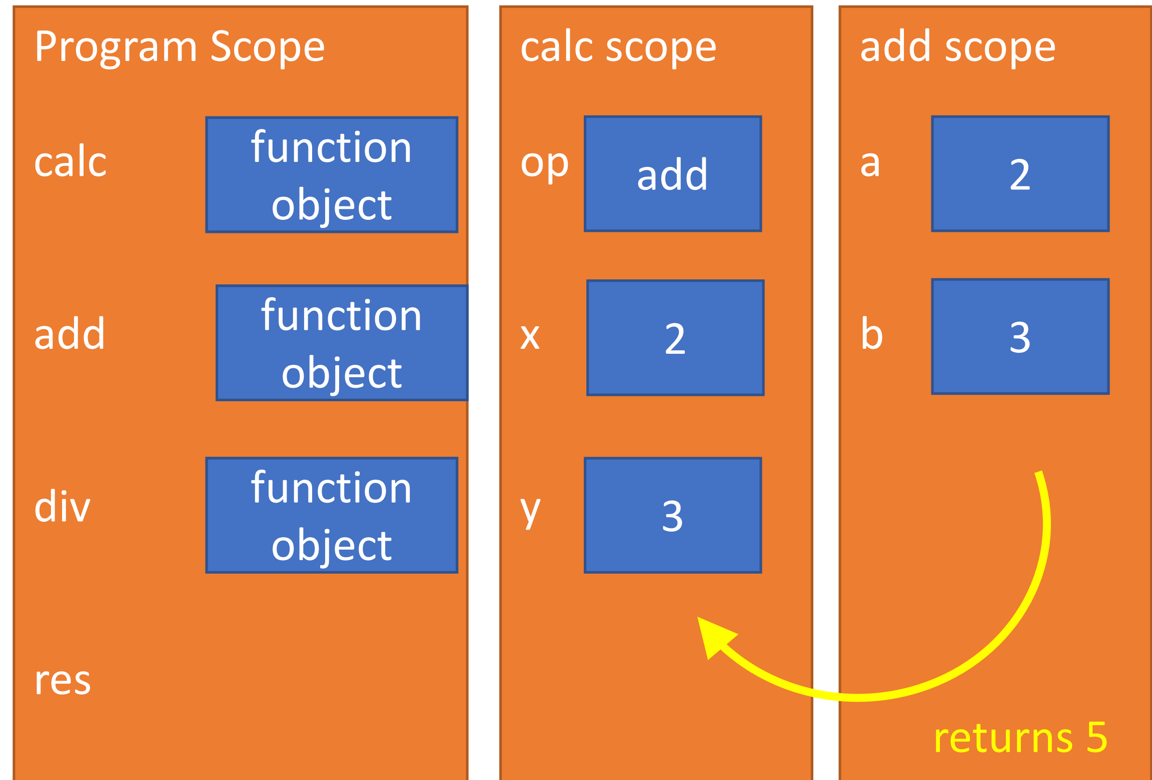
```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```

Function call in calc scope: add with formal params a=2 and b=3



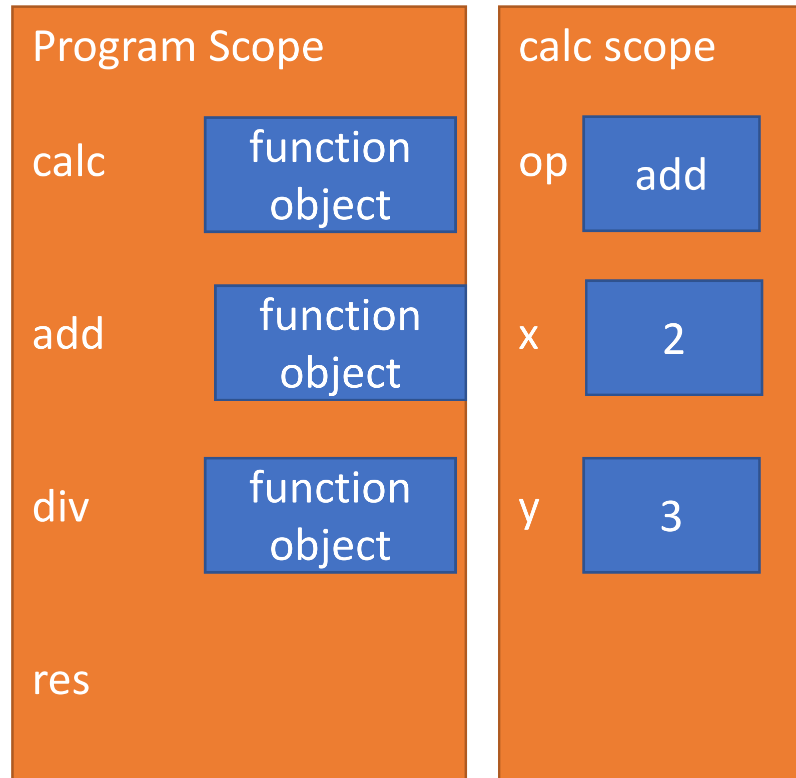
EXECUTE LINE OF add

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



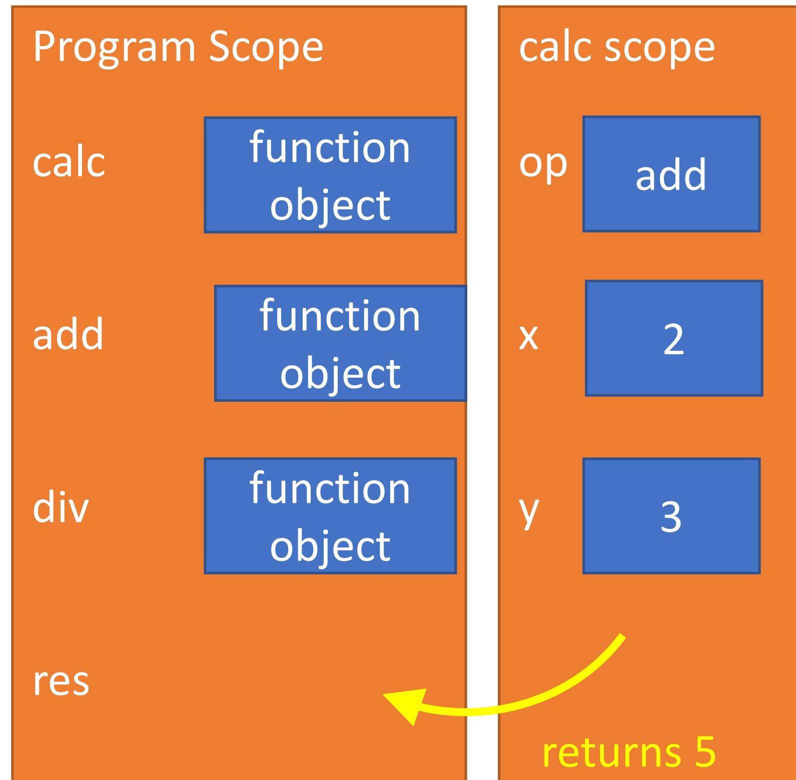
REPLACE FUNC CALL WITH RETURN

```
def calc(op, x, y):
    return op(x,y)    5

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



EXECUTE LINE OF calc

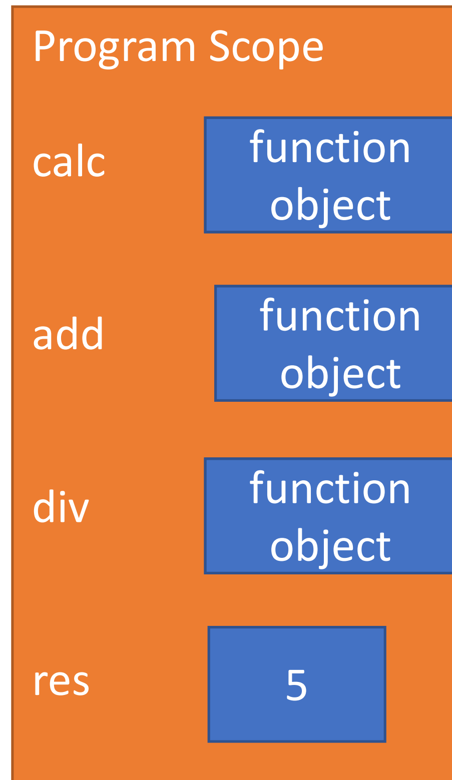
```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



REPLACE FUNC CALL WITH RETURN

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```

5



YOU TRY IT!

- Do a similar trace with the function call

```
def calc(op, x, y):
    return op(x,y)

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(div,2,0)
```

What is the value of res and what gets printed?

ANOTHER EXAMPLE: FUNCTIONS AS PARAMS

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```

call func_a, takes no parameters

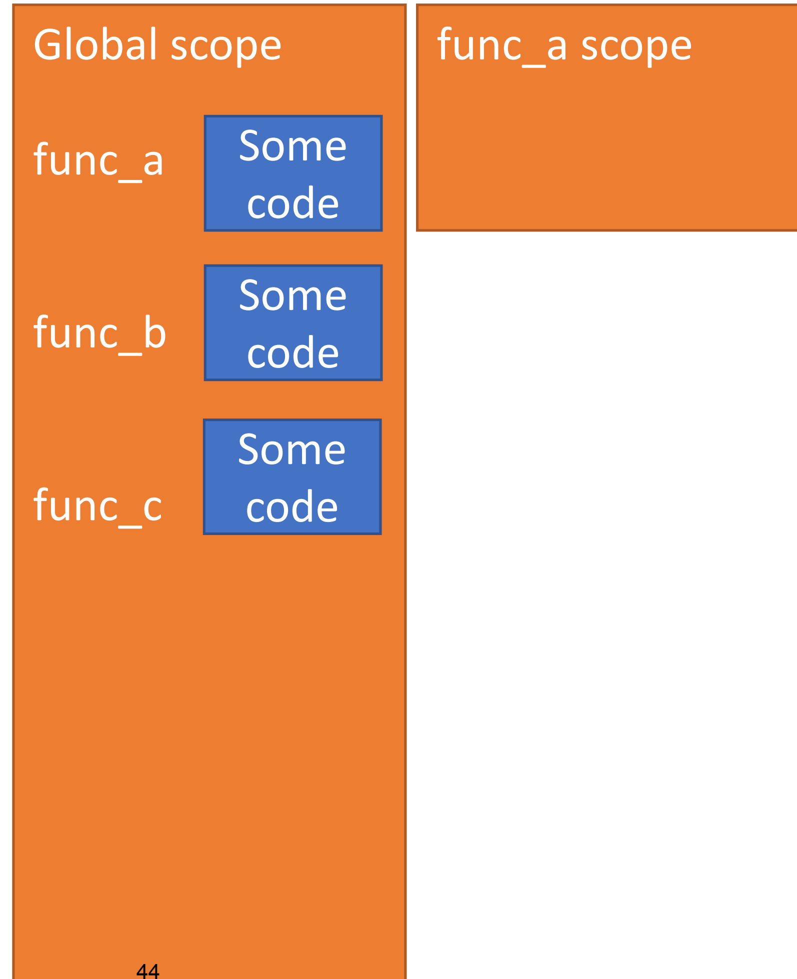
call func_b, takes one parameter, an int

call func_c, takes two parameters, another function and an int

FUNCTIONS AS PARAMETERS

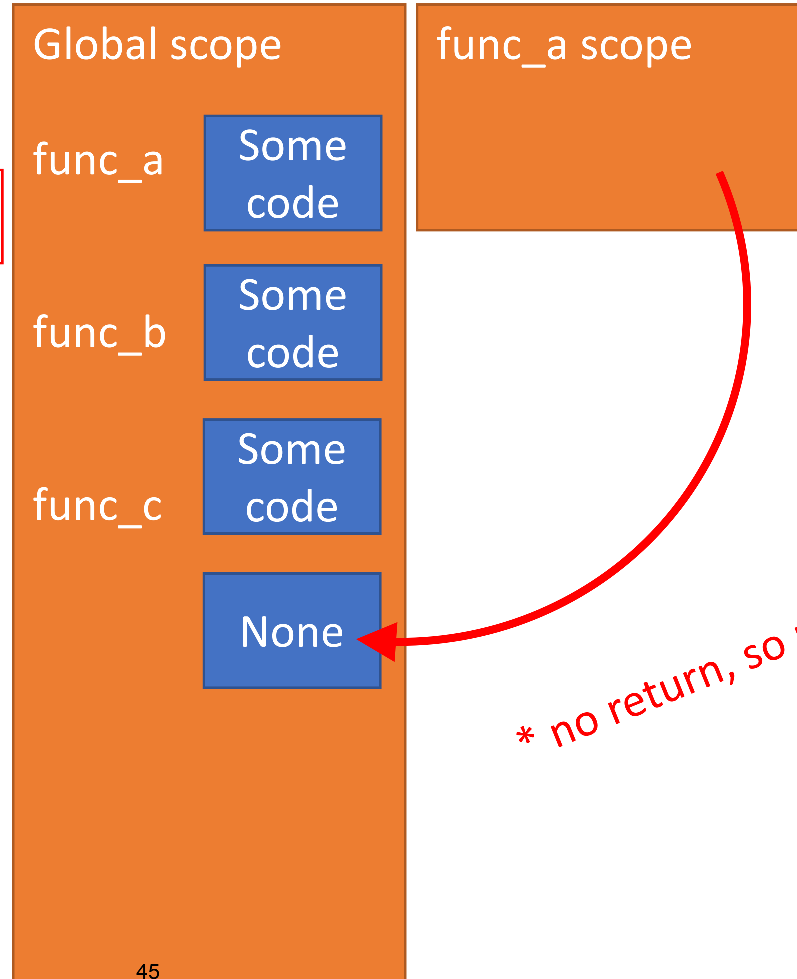
No bindings (no parameters)

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



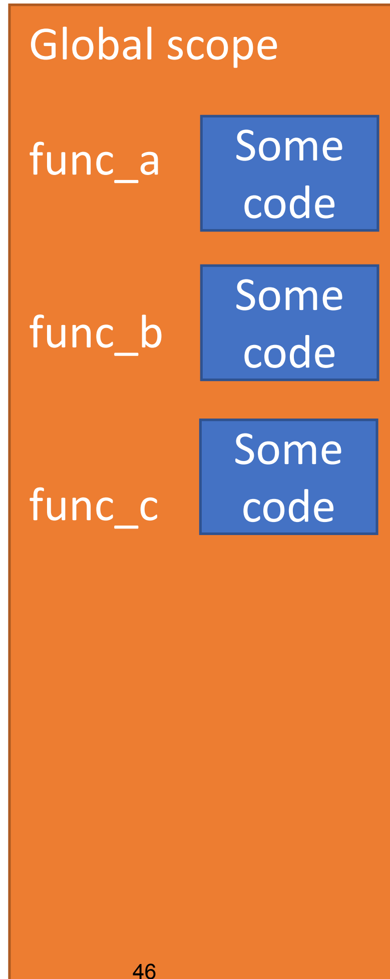
FUNCTIONS AS PARAMETERS

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



FUNCTIONS AS PARAMETERS

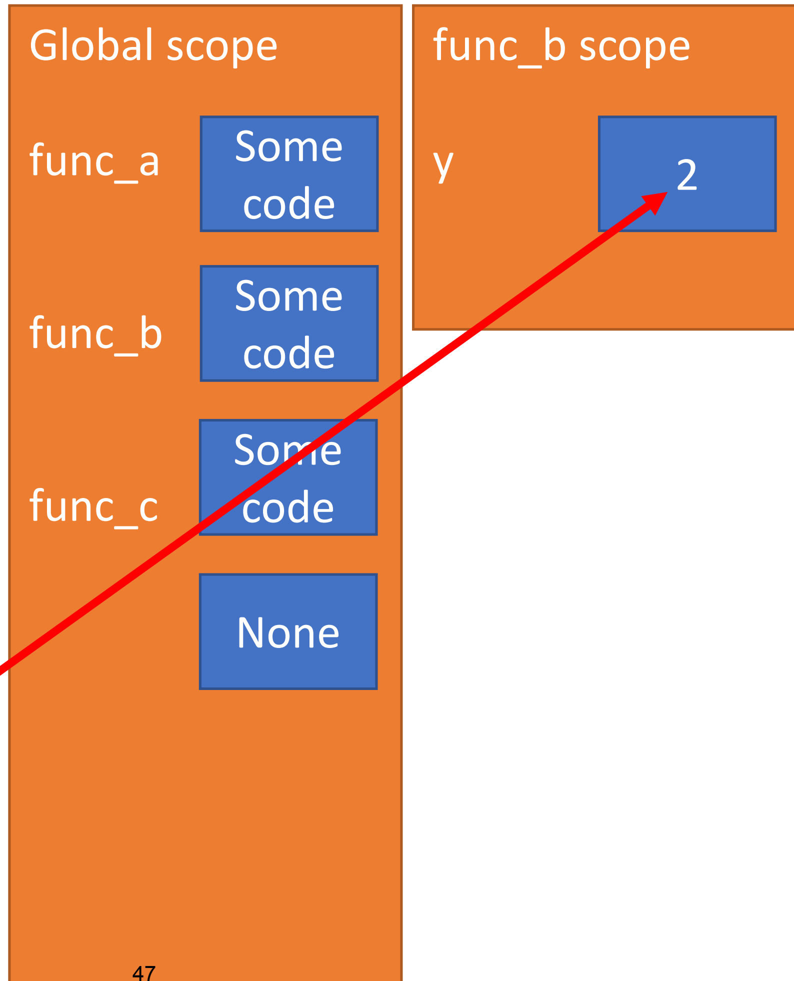
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



print displays None on console

FUNCTIONS AS PARAMETERS

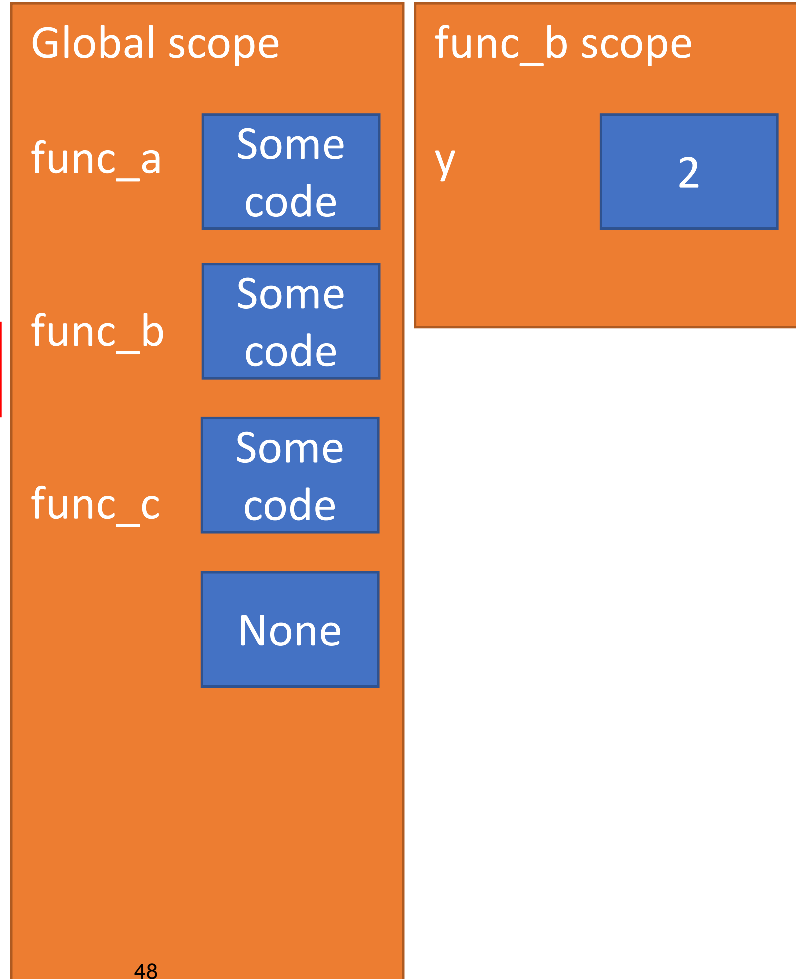
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



FUNCTIONS AS PARAMETERS

body prints 'inside func_b' on console

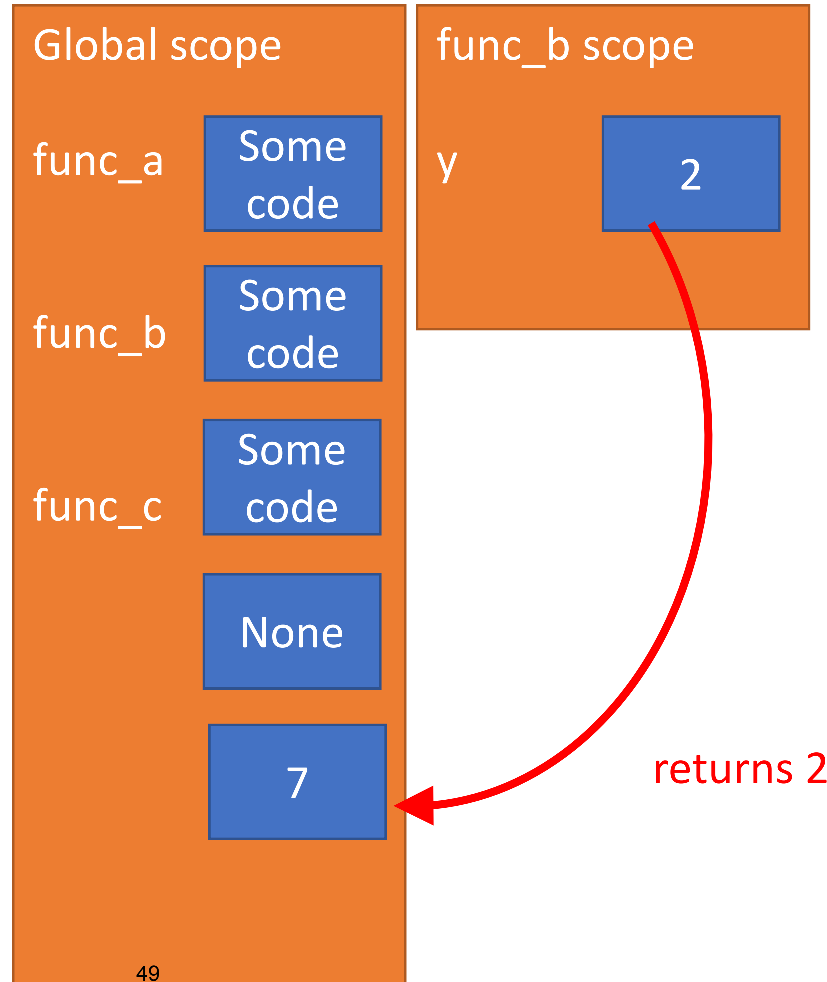
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



FUNCTIONS AS PARAMETERS

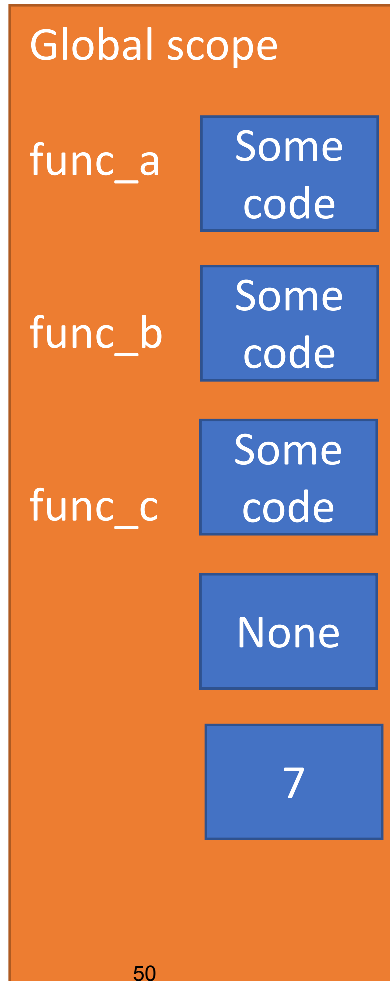
value of 2 is returned and added to 5

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



FUNCTIONS AS PARAMETERS

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```

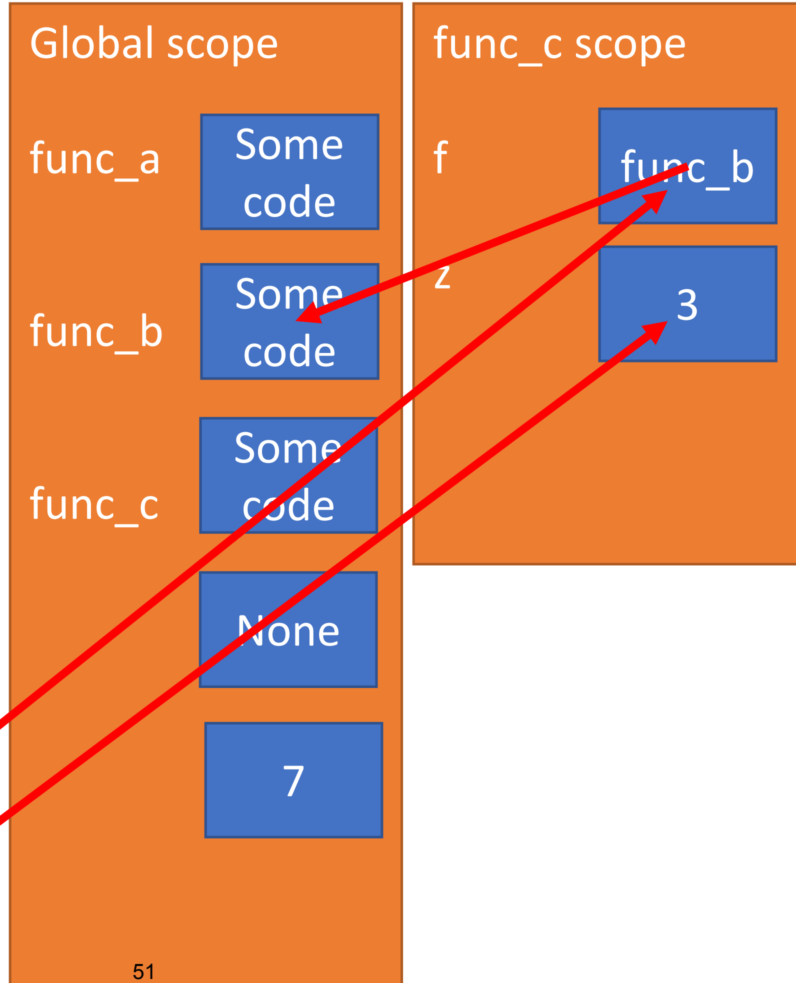


print displays 7 on console

FUNCTIONS AS PARAMETERS

body of func_c prints
'inside func_c' on console

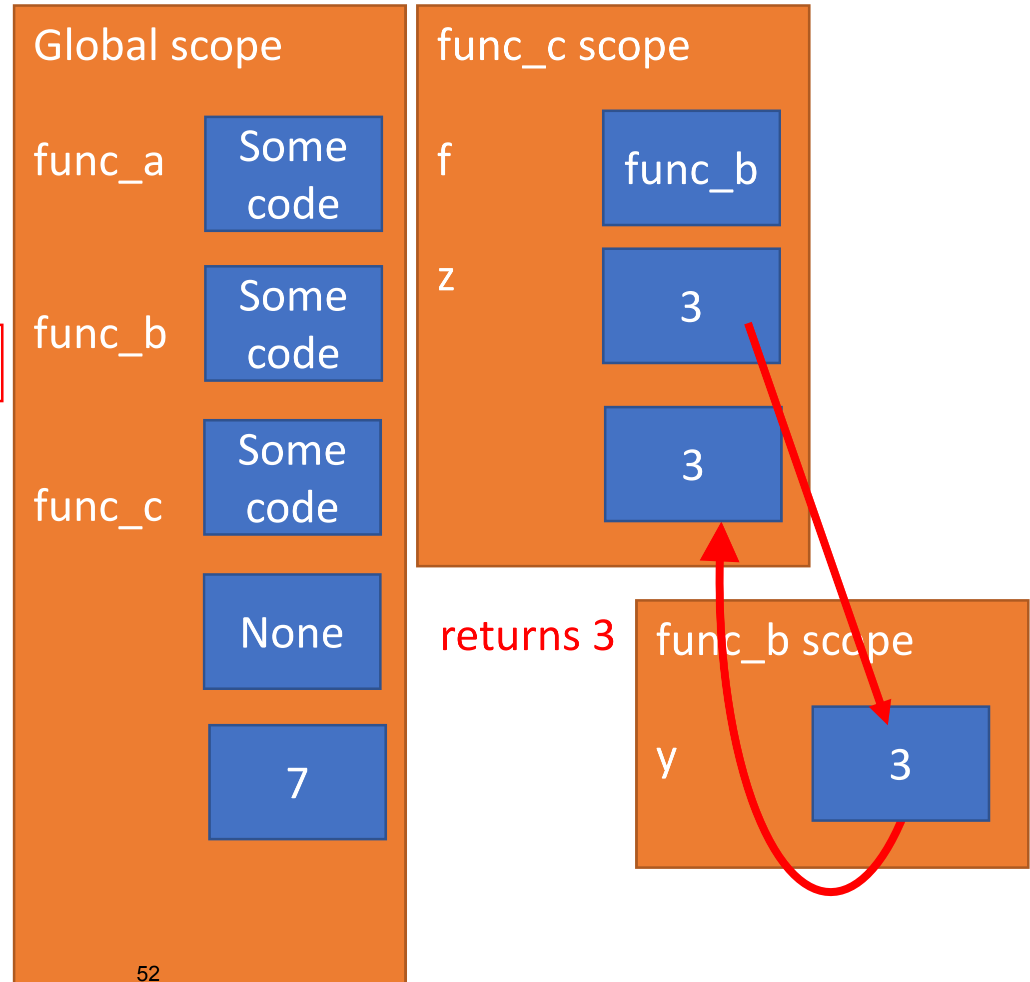
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



FUNCTIONS AS PARAMETERS

body of func_b prints 'inside func_b' on console

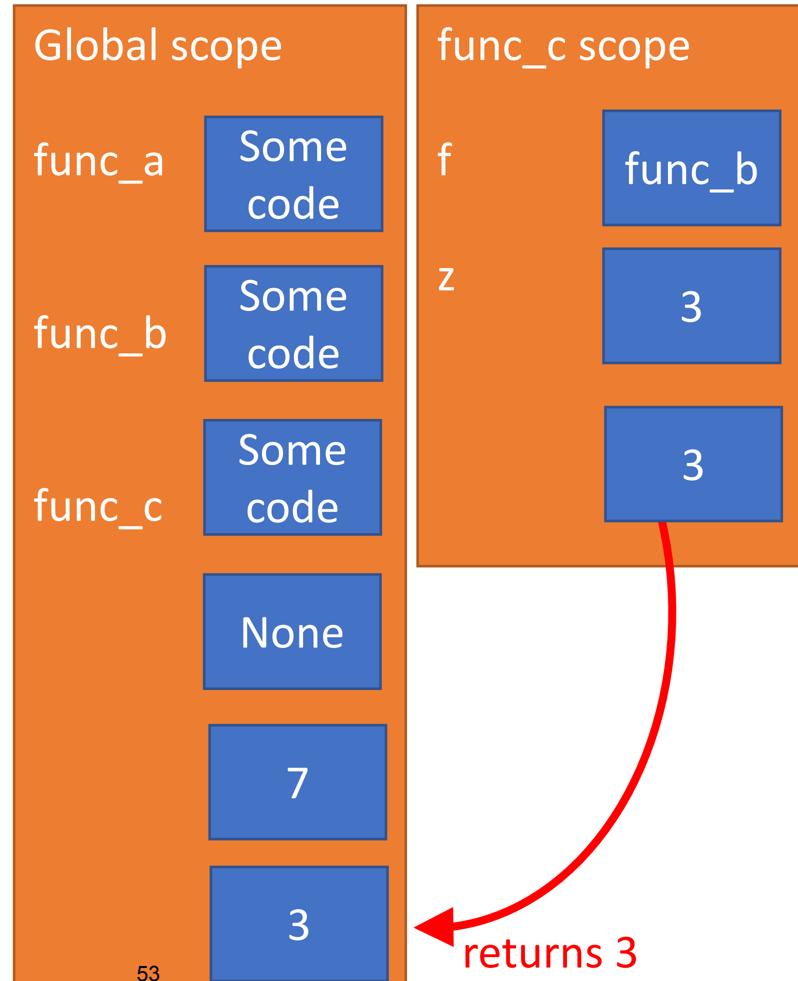
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)    3
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



FUNCTIONS AS PARAMETERS

print displays 3 on console

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



YOU TRY IT!

- Write a function that meets these specs.

```
def apply(criteria,n):
    """
    * criteria is a func that takes in a number and returns a bool
    * n is an int
    Returns how many ints from 0 to n (inclusive) match
    the criteria (i.e. return True when run with criteria)
    """
```

SUMMARY

- Functions are first class objects
  - They have a **type**
  - They can be **assigned as a value** bound to a name
  - They can be used as an **argument** to another procedure
  - They can be **returned** as a value from another procedure
- Have to be careful about environments
  - Main program runs in the global environment
  - Function calls each get a new temporary environment
- This enables the creation of concise, easily read code

MITOpenCourseWare
https://ocw.mit.edu

6.100L Introduction to Computer Science and Programming Using Python
Fall 2022

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