DECOMPOSITION, ABSTRACTION, FUNCTIONS

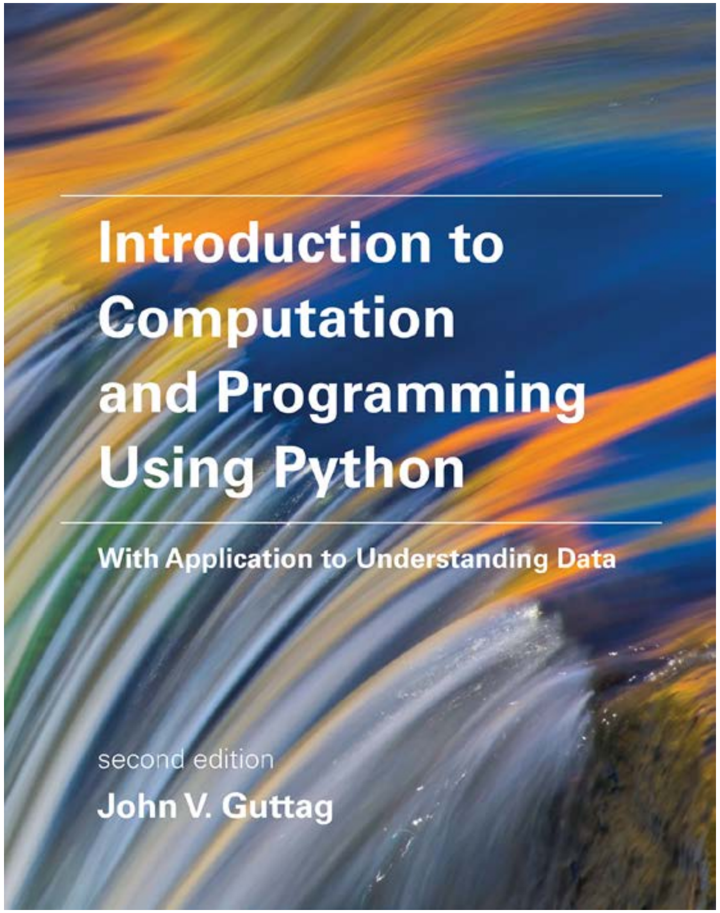
(download slides and .py files from Stellar to follow along)

6.0001 LECTURE 4

ANA BELL

Assigned Reading

- Sections 4.1 – 4.2
- For next lecture
  - 4.3 – 4.6
  - 5.1 – 5.5


Introduction to Computation and Programming Using Python
With Application to Understanding Data
second edition
John V. Guttag

https://mitpress.mit.edu/sites/default/files/Guttag_errata_revised_083117.pdf

TODAY

- Structuring programs and hiding details
- Functions
- Specifications
- Scope
- **Microquiz**
  - Adjourn class at by 4:00
  - Students with accommodations go to 26-142
  - Quiz starts at 4:05
  - Two programming problems

LEARNING TO PRODUCE CODE

- So far covered language mechanisms
  - Syntax, variables
  - Branching (if/elif/else)
  - Loops (for, while)
- Know everything you need to know to accomplish anything that can be accomplished by computation
- But there are two other most important concepts in programming

DECOMPOSITION AND ABSTRACTION

- **Decomposition** is about dividing a program into self-contained parts that can be combined to solve the problem
  - Ideally can be reused
- **Abstraction** is all about ignoring unnecessary detail
  - Used to separate **what** something does, from **how** it does it

AN EXAMPLE: THE SMART PHONE

- A black box
- Don’t know the details of how it works
- Do know the user interface
- Somehow converts a sequence of screen touches and sounds into useful functionality
- **Abstraction**:

  We don’t need to know
  how it works
  to know
  how to use it

ABSTRACTION ENABLES DECOMPOSITION

- 100’s of distinct parts
- Designed and manufactured by 10’s of companies
  - Many of which do not communicate with each other
- **Decomposition**:
  Each component maker has to know **how its component interfaces** to other components, but **not how other components are implemented**

SUPRESS DETAILS WITH ABSTRACTION

- Think of a code module as a **black box**
  - Cannot see implementation details
  - Do not need to see details
  - Do not want to see details

- Achieve abstraction with **function specifications** using **docstrings**

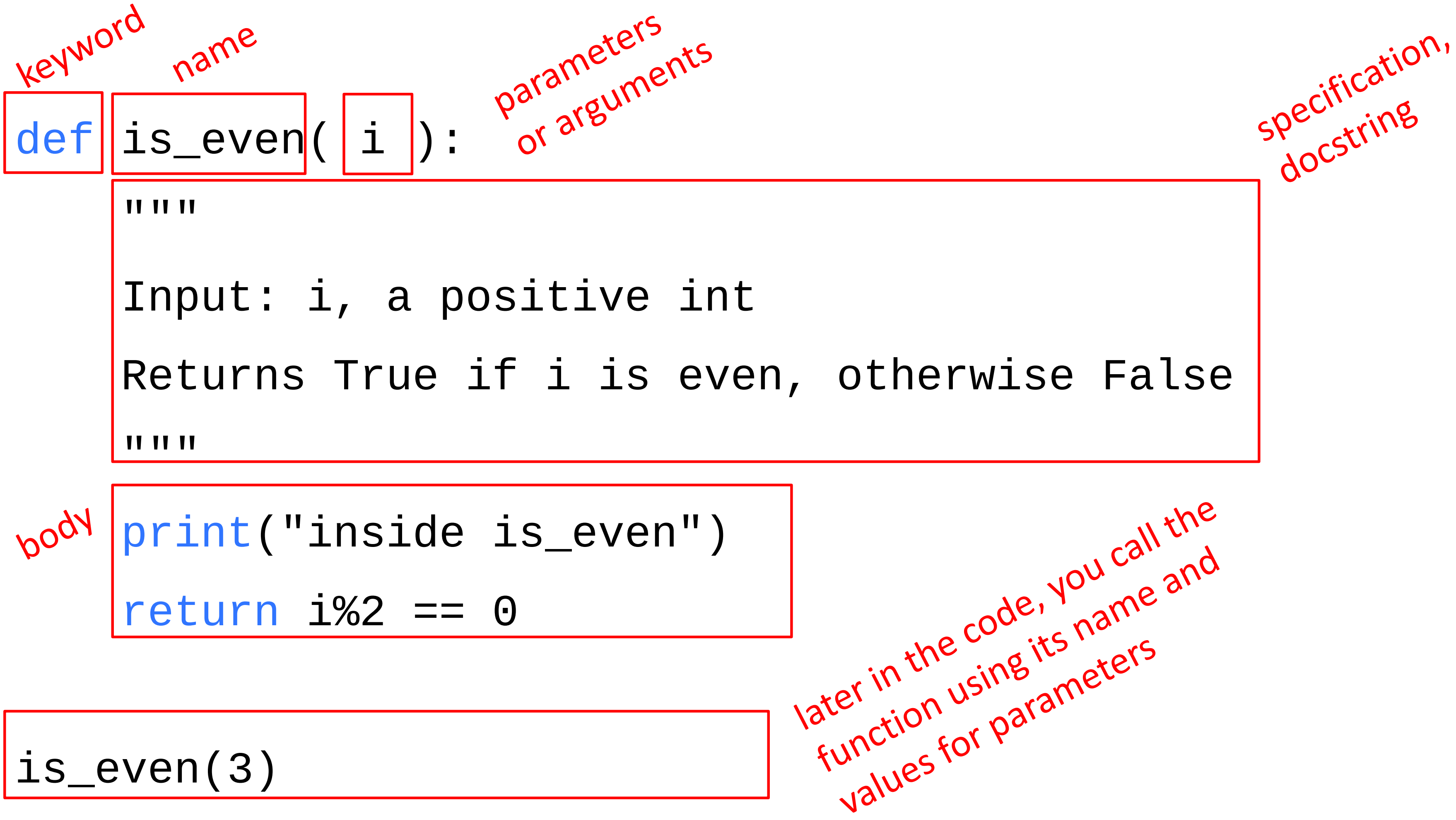
CREATE STRUCTURE WITH DECOMPOSITION

- Divide code into **modules**
  - Are **self-contained**
  - Used to **break up** code
  - Intended to be **reusable**
  - Keep code **organized**
  - Keep code **coherent**
- This lecture, decomposition with **functions**
- In a few weeks, decomposition with **classes**

FUNCTIONS IN PYTHON

- Function characteristics:
  - Has a **name**
  - Has (formal) **parameters** (0 or more)
  - Has a **docstring** (optional but recommended)
    - A comment delineated by """ (triple quotes) that provides a **specification** for the function
  - Has a **body**
  - **Returns** something
- Functions are not run in a program until they are “**called**” or “**invoked**” in a program

HOW TO WRITE and CALL/INVOKE A FUNCTION



keyword

name

parameters or arguments

specification, docstring

body

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
```

run some commands

keyword

expression to evaluate and return

VARIABLE SCOPING

- **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

```
a = 3
z = f( a )
```

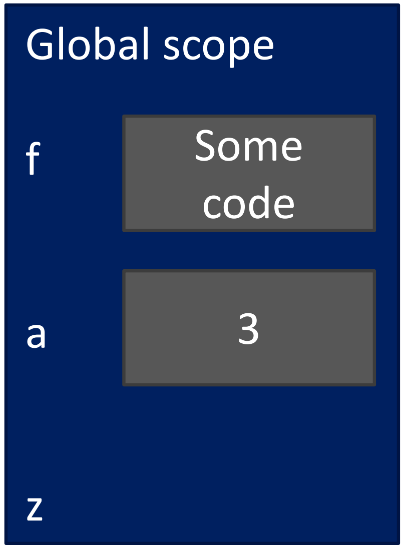
actual parameter

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

VARIABLE SCOPE

After executing 1st assignment

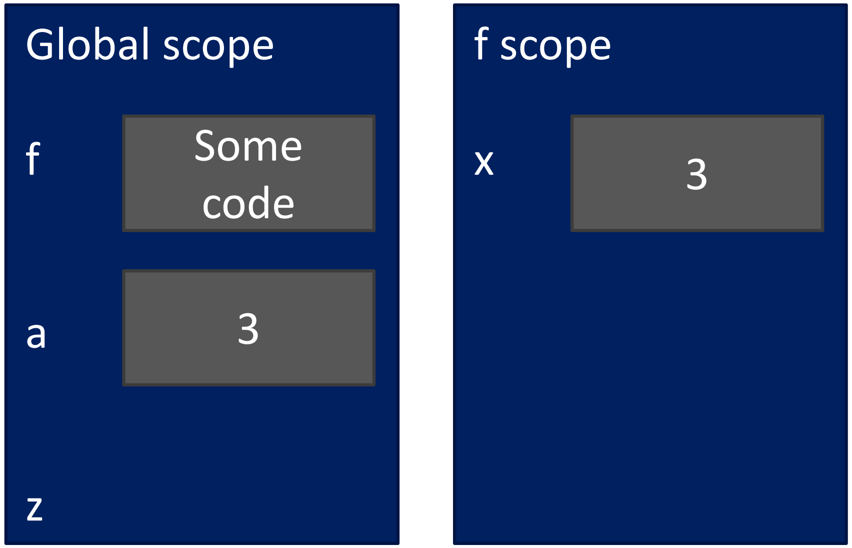
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

a = 3
z = f( a )
```



VARIABLE SCOPE

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

a = 3
z = f( a )
```

After f invoked



Global scope

f

Some code

a

3

z

f scope

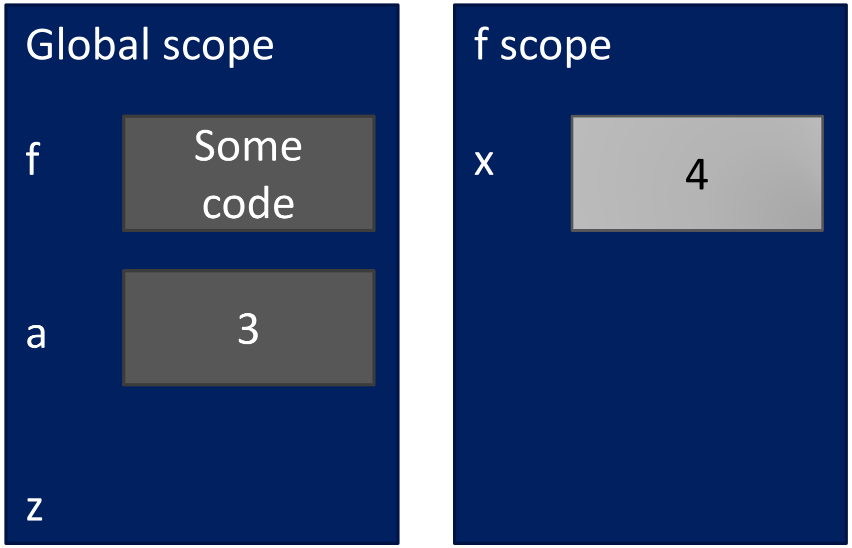
x

3

VARIABLE SCOPE

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

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

Just before f returns



Global scope

f Some code

a 3

z

f scope

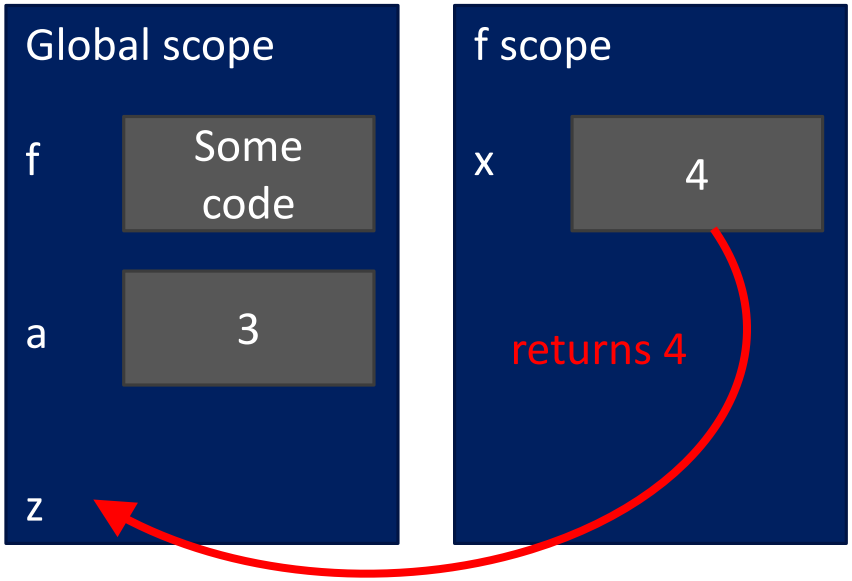
x 4

VARIABLE SCOPE

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

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

Function call replaced with what is returned

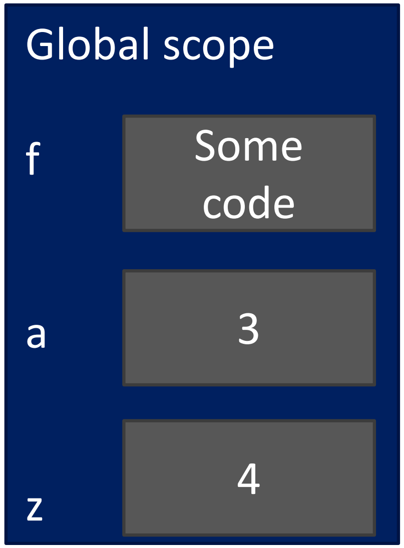
During the return



VARIABLE SCOPE

After executing 2nd assignment

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

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



Global scope

f	Some code
a	3
z	4

WHAT IF THERE IS NO return

```
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
- No static semantic error generated

☆ LIVE EXERCISE

MORE ON return

- Return only has meaning **inside** a function
- Only **one** return executed per function invocation
  - But code can contain multiple return statements
- No code within the function executed after the return is exectuted
- Has a value associated with it, **given to function caller**

FUNCTIONS AS PARAMETERS

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

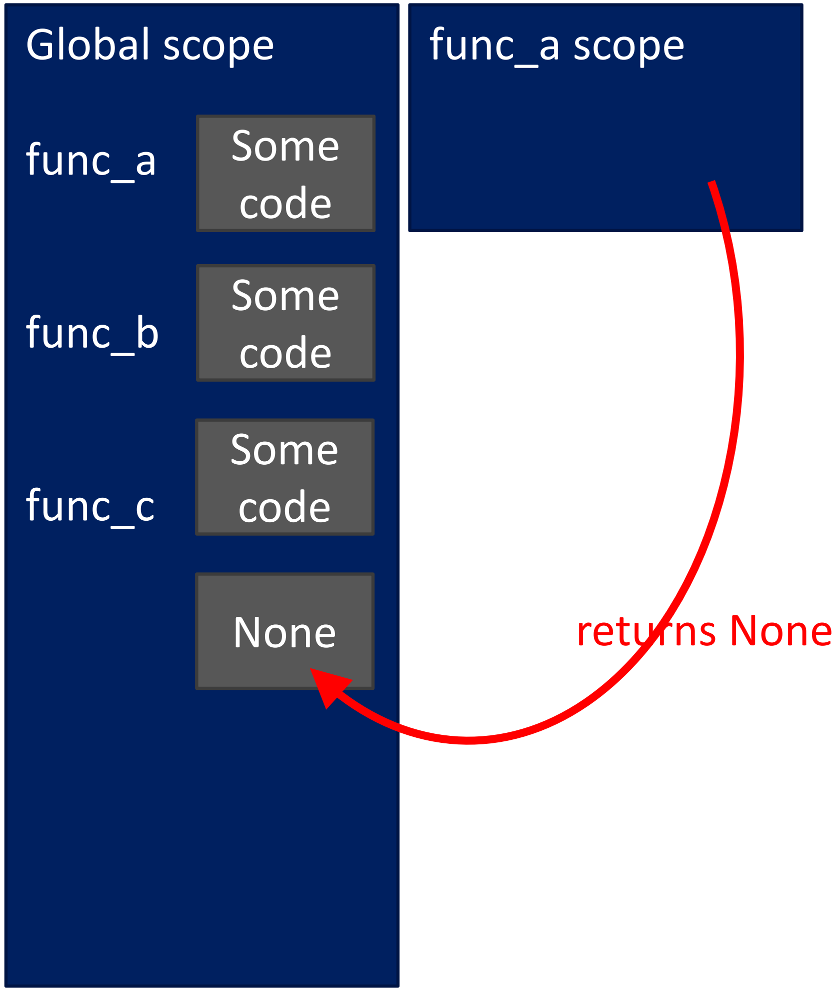
call `func_a`, takes no parameters

call `func_b`, takes one parameter

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

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))
```



Global scope

func_a Some code

func_b Some code

func_c Some code

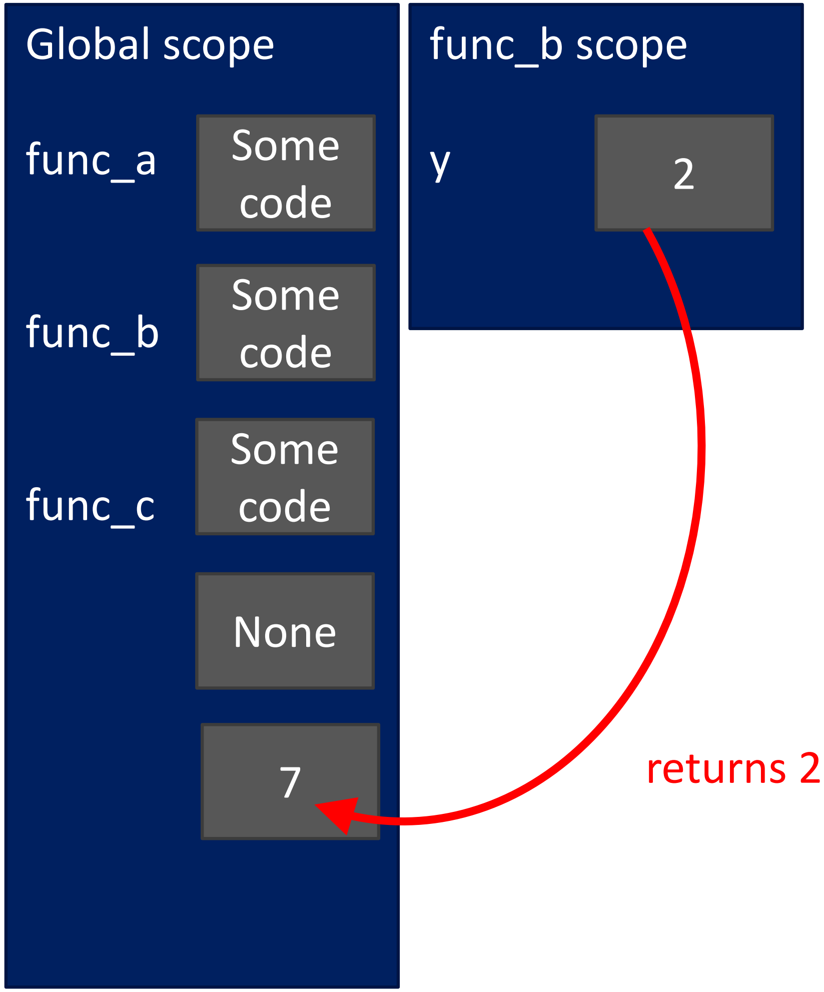
None

func_a scope

returns None

FUNCTIONS AS PARAMETERS

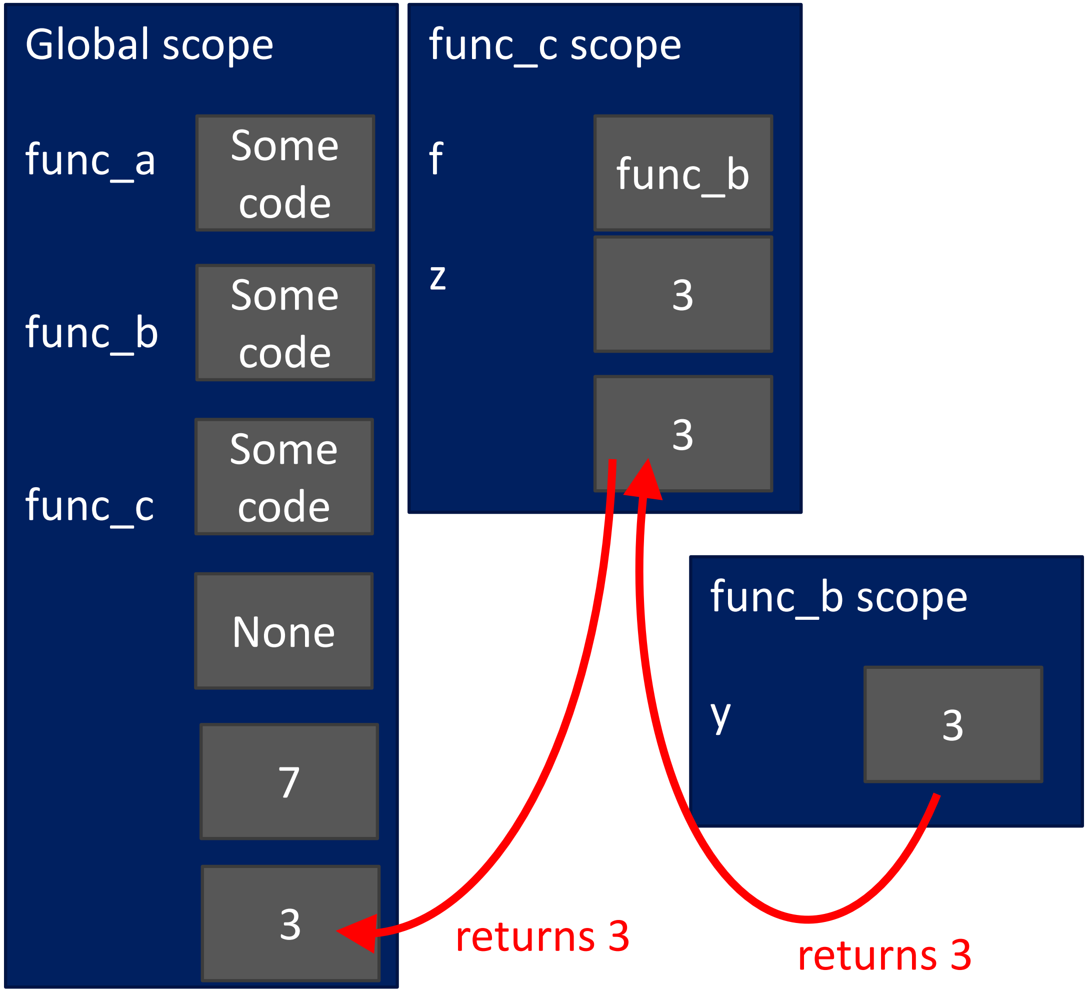
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)
print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



FUNCTIONS AS PARAMETERS

☆ LIVE EXERCISE

```
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))
```



Global scope

func_a Some code

func_b Some code

func_c Some code

None

7

3

func_c scope

f func_b

z 3

3

func_b scope

y 3

returns 3

returns 3

VARIABLE SCOPING

- Inside a function, **can access** a variable defined outside
- Inside a function, **cannot modify** a variable defined outside

```
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 in which function *g* is defined

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

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

UnboundLocalError: local variable 'x' referenced before assignment

VARIABLE SCOPING

- Inside a function, **can access** a variable defined outside
- Inside a function, **cannot modify** a variable defined outside

```
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!

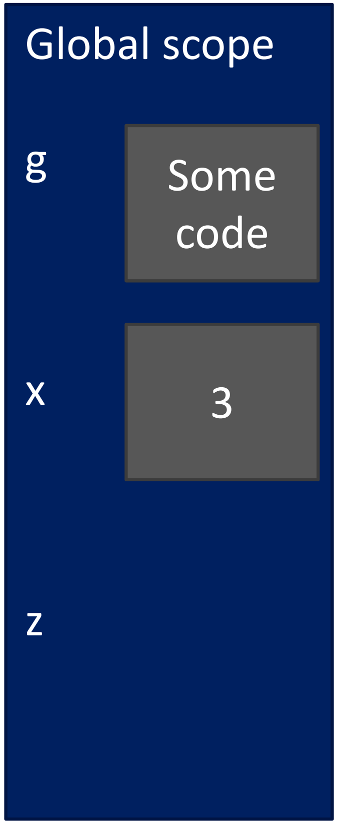
Python Tutor is your best friend to help sort this out!

http://www.pythontutor.com/

SCOPE DETAILS

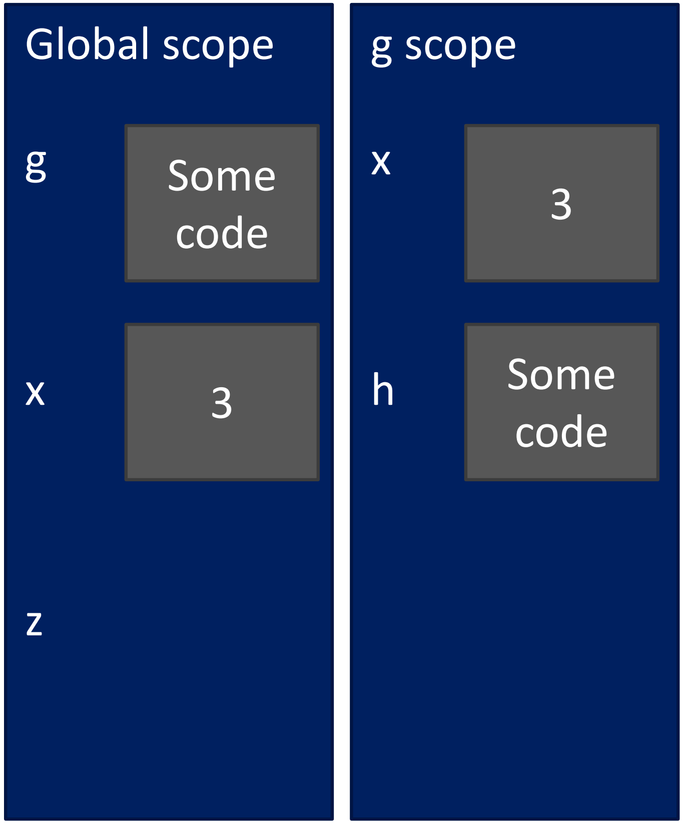
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    print(h())
    return x

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

Some code



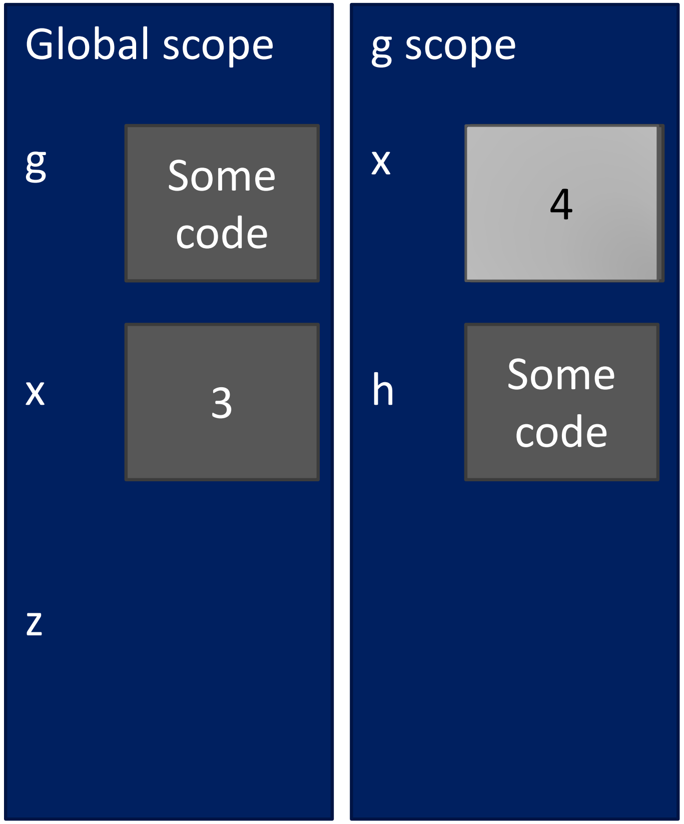
SCOPE DETAILS

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

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



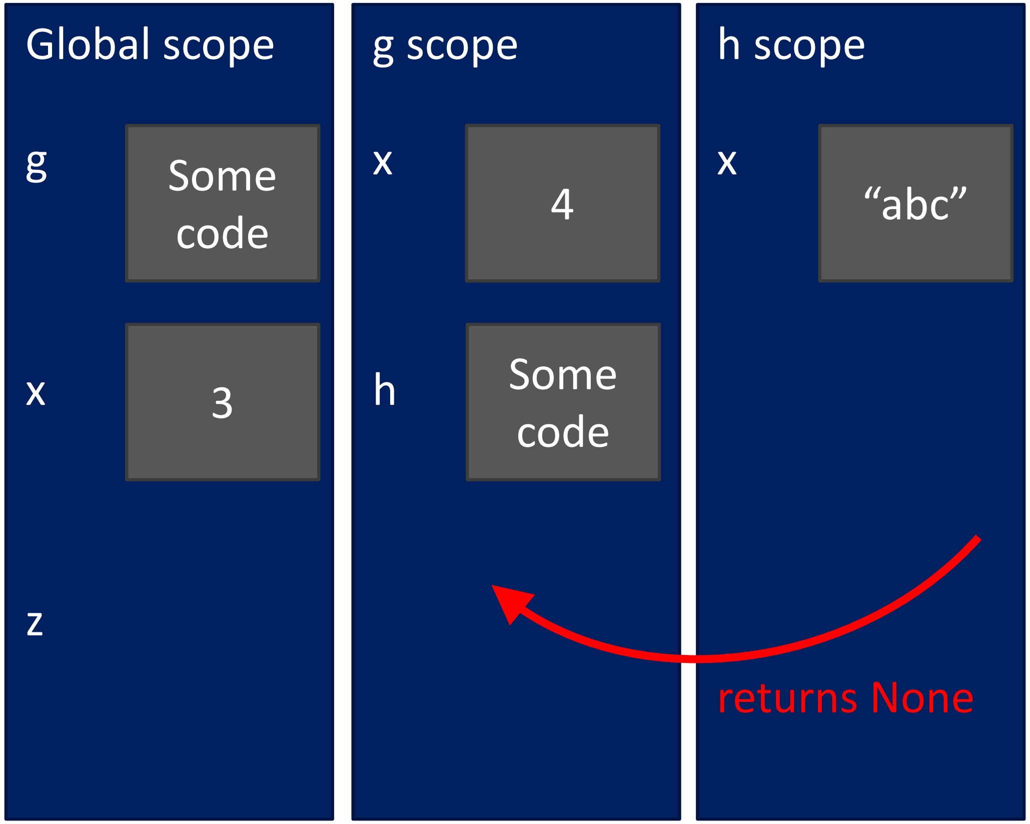
SCOPE DETAILS

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

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



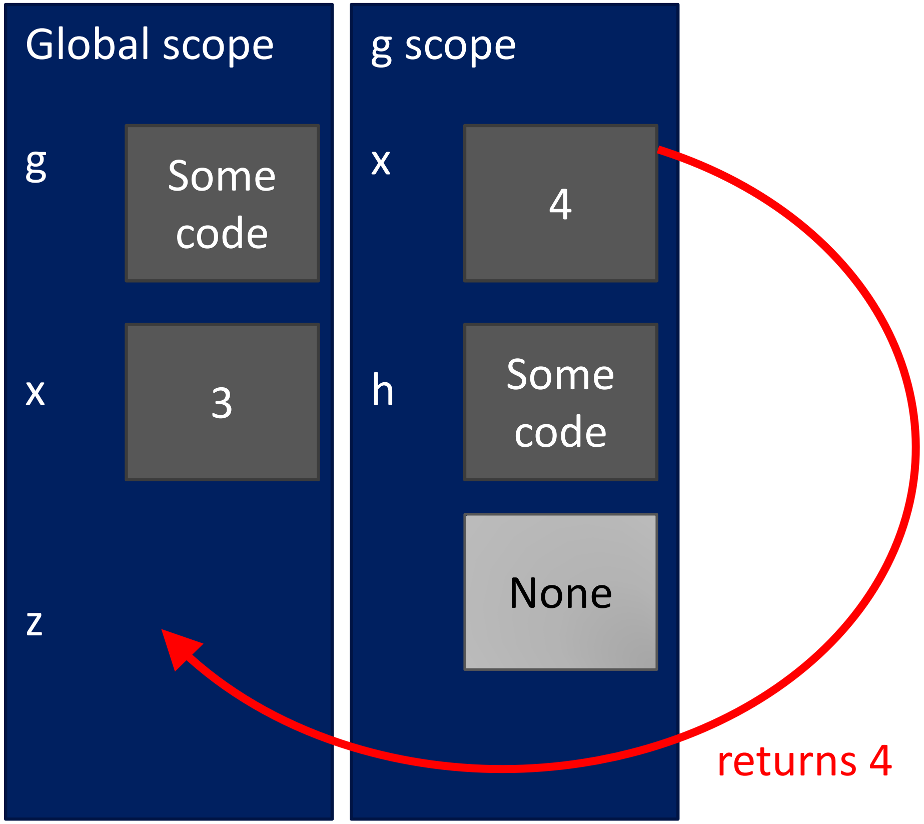
SCOPE DETAILS

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

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



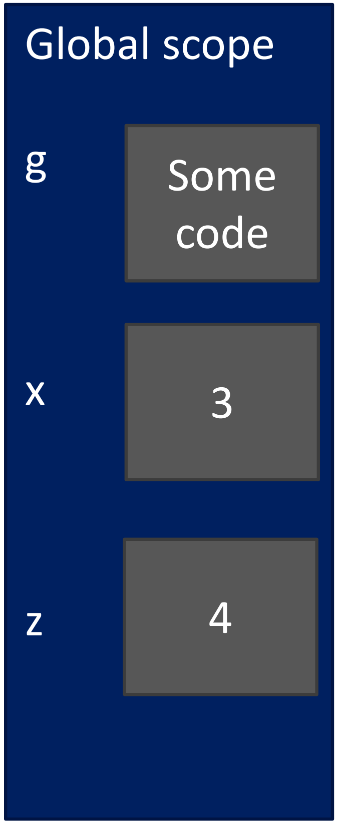
SCOPE DETAILS

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

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



SCOPE DETAILS

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

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



Global scope

g   Some code

x   3

z   4

DECOMPOSITION & ABSTRACTION

- Powerful together
- Code can be used many times but only has to be debugged once!