

CSCI 145 · Introduction to Computing · Fall 2026

Scope & the Call Stack

Lecture 3 — global and local scope, tracing execution with stack diagrams

Outline

- Review of functions
-

- Scope
-

- Tracing the call stack
-

Part 1 of 3

Review of functions

Function syntax, a reminder

```
def function_name(param1, param2):  
    """  
    One or two sentences describing what the function does.  
  
    Params:  description and types of parameters  
    Returns: description and type of the returned value (if any)  
    """  
    # do something in your function  
    # to calculate outputs  
    result = 0  
  
    return result # return a value if necessary
```

Example 1

```
def greet(name):  
    """Greet someone."""  
    print("Hello " + name)  
    print("How are you?")
```



We've added a parameter we can then use in the function

```
>>> greet("Vicki")  
Hello Vicki  
How are you?
```



When we call the function, we set the value of the parameter

Example 2

```
def double(x):  
    """  
    Double the input.  
  
    Params:  x (int/float/str) -- value to be doubled  
    Returns: doubled value (int/float/str)  
    """  
    return x * 2
```

```
>>> double(5) + 1  
11
```

← The returned value becomes the value of the function call

```
>>> double(6.7)  
13.4
```

Part 2 of 3

Scope

The `double` function, revisited

```
def double(x):  
    """Double the input."""  
    doubled_value = x * 2  
    return doubled_value
```

```
>>> double(5)  
10
```

```
>>> doubled_value  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'doubled_value' is not defined
```

← What does this do?

← To understand this, we need to understand scope

Scope

- The **scope** of a variable is the region of the code where it can be accessed
- Functions create **local scope** — variables created inside are local and cannot be accessed from outside the function
- Variables created outside of any function are in **global scope**, and can be accessed anywhere

Scope — example 1

LOCAL SCOPE

```
def double(x):  
    doubled_value = x * 2
```

GLOBAL SCOPE

```
>>> double(5)  
>>> doubled_value
```



doubled_value only exists inside double

Scope — example 2

What happens when we run this code?

LOCAL SCOPE

```
def increment(x):  
    result = x + 1  
    return result
```

GLOBAL SCOPE

```
y = 6  
z = increment(y)
```

main

y: 6

z: ? 7

increment

x: 6

result: 7

Variables don't leak out of their scope

```
def increment(x):  
    result = x + 1  
    return result
```

```
y = 6  
z = increment(y)  
print(result)
```

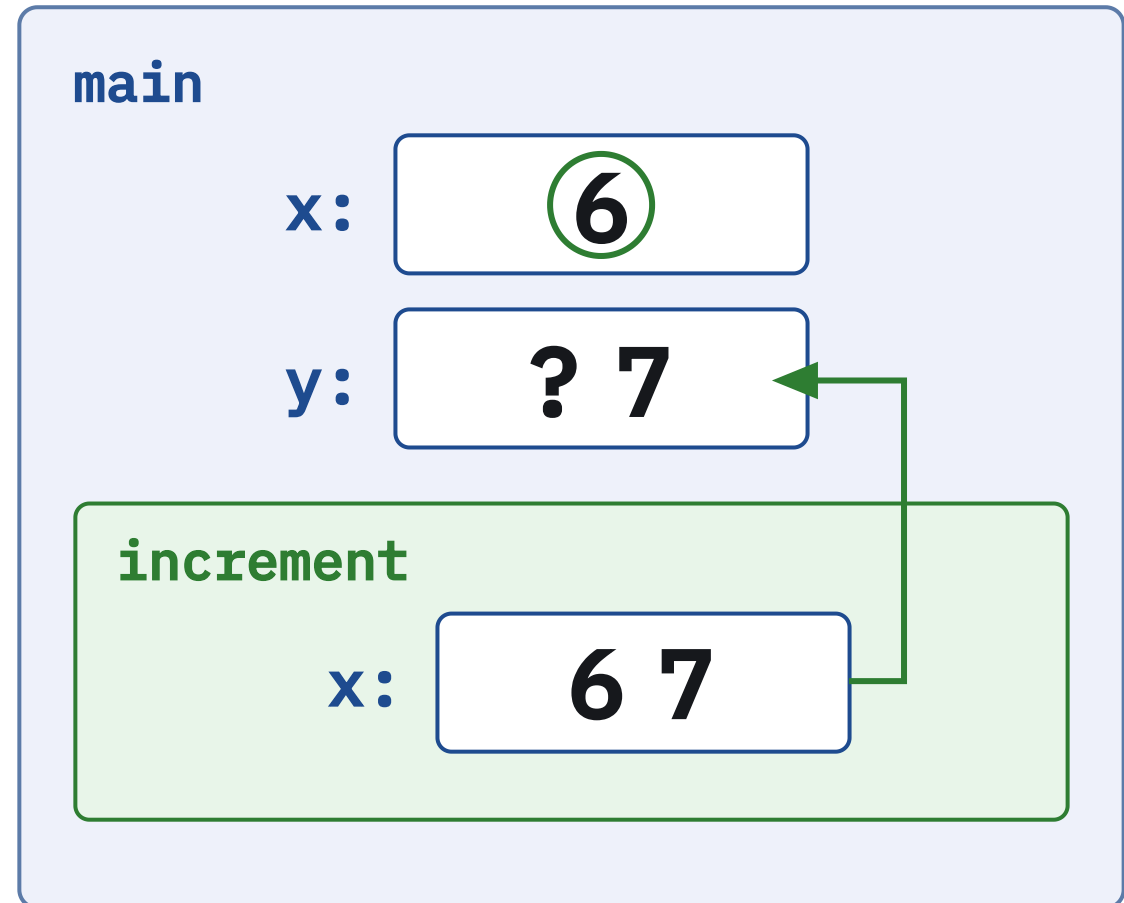
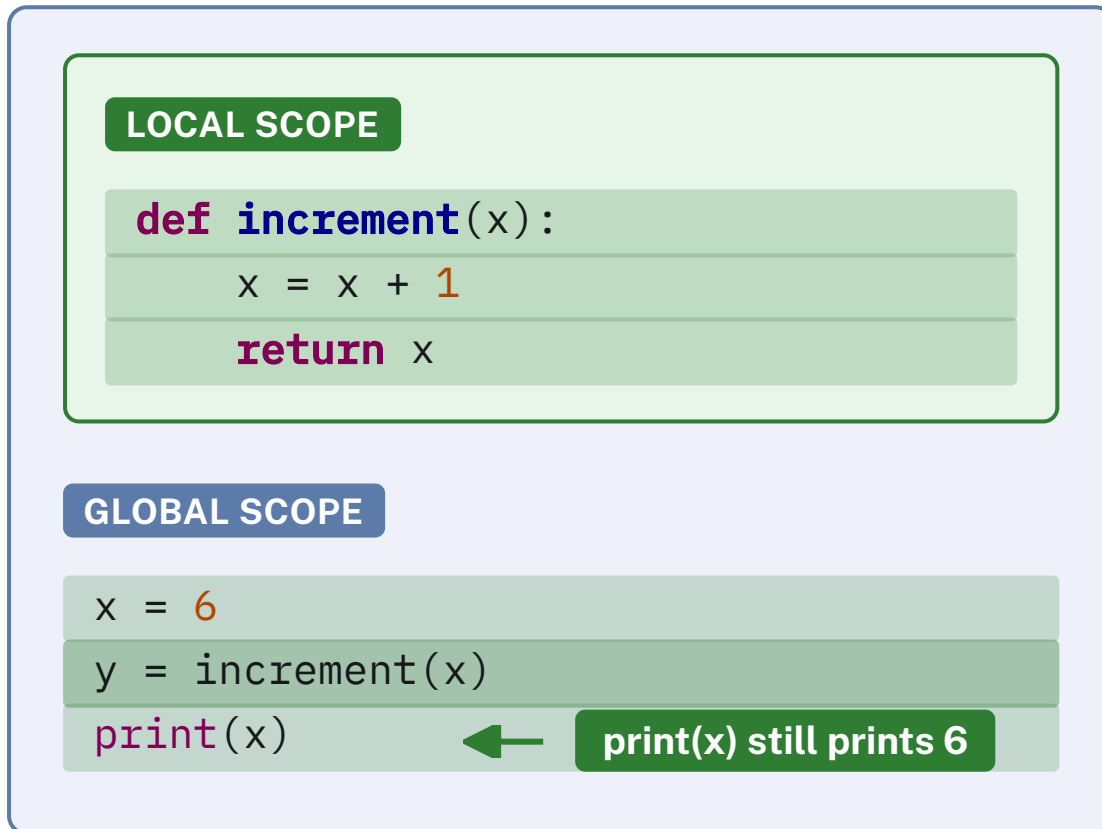
NameError: name 'result' is not defined



result only exists inside increment

Same name, different scope

What happens when we use the same variable name in both scopes?



What is the value of `x` in the global scope after this code runs?

```
def mystery(x):  
    x = x * 3  
    return x  
  
x = 4  
result = mystery(x)
```

Go to **slido.com** and enter the code:

3674841

Or [join directly here](#)



What is the value of `x` in the global scope after this code runs?

LOCAL SCOPE

```
def mystery(x):  
    x = x * 3  
    return x
```

GLOBAL SCOPE

```
x = 4  
result = mystery(x)  
print(x) ← print(x) still prints 4
```

main

x:

4

result:

? 12

mystery

x:

4 12

What is the value of `y` in the global scope after this code runs?

```
y = 0

def calculate(x, y):
    a = x
    b = y + 2
    c = a + b + 3
    return c

y = calculate(3, 2)
```

Go to **slido.com** and enter the code:

3674841

Or [join directly here](#)



What is the value of `y` in the global scope after this code runs?

LOCAL SCOPE

```
def calculate(x, y):  
    a = x  
    b = y + 2  
    c = a + b + 3  
    return c
```

GLOBAL SCOPE

```
y = 0  
y = calculate(3, 2) ← y is now 10
```

main

y: 0 10

calculate

x: 3

y: 2

a: 3

b: 4

c: 10

Part 3 of 3

Tracing the call stack

Tracing the call stack

- Every time a function is called, Python pushes a new **frame** onto the **call stack** — a fresh local scope for that specific call.
- When the function returns, its frame is **popped**, and execution picks back up right where it left off — with the returned value in hand.

Stack diagrams

```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```

Stack diagrams

```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```

main z: 5

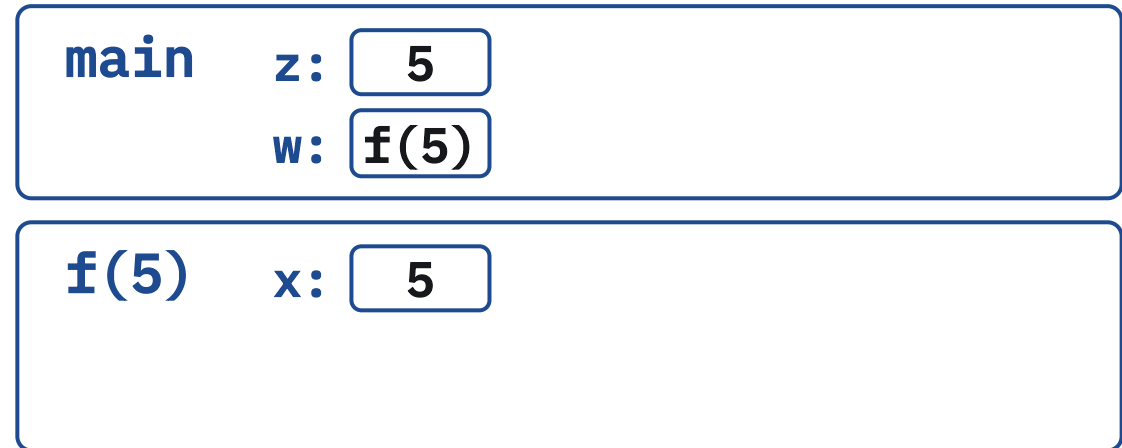
Stack diagrams

```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```

main	z:	5
	w:	f(5)

Stack diagrams

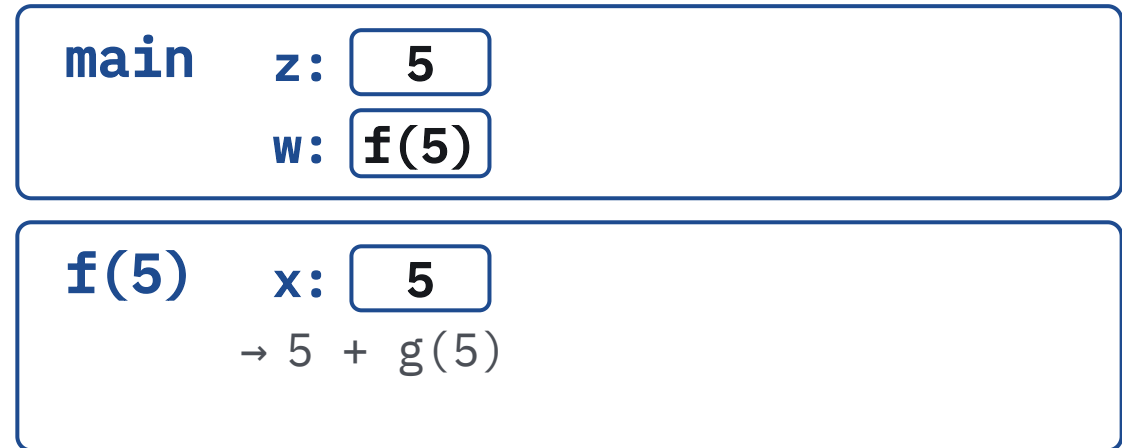
```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```



Calling `f(z)` pushes a new frame — a local scope for this call, with its own copy of the parameter `x`.

Stack diagrams

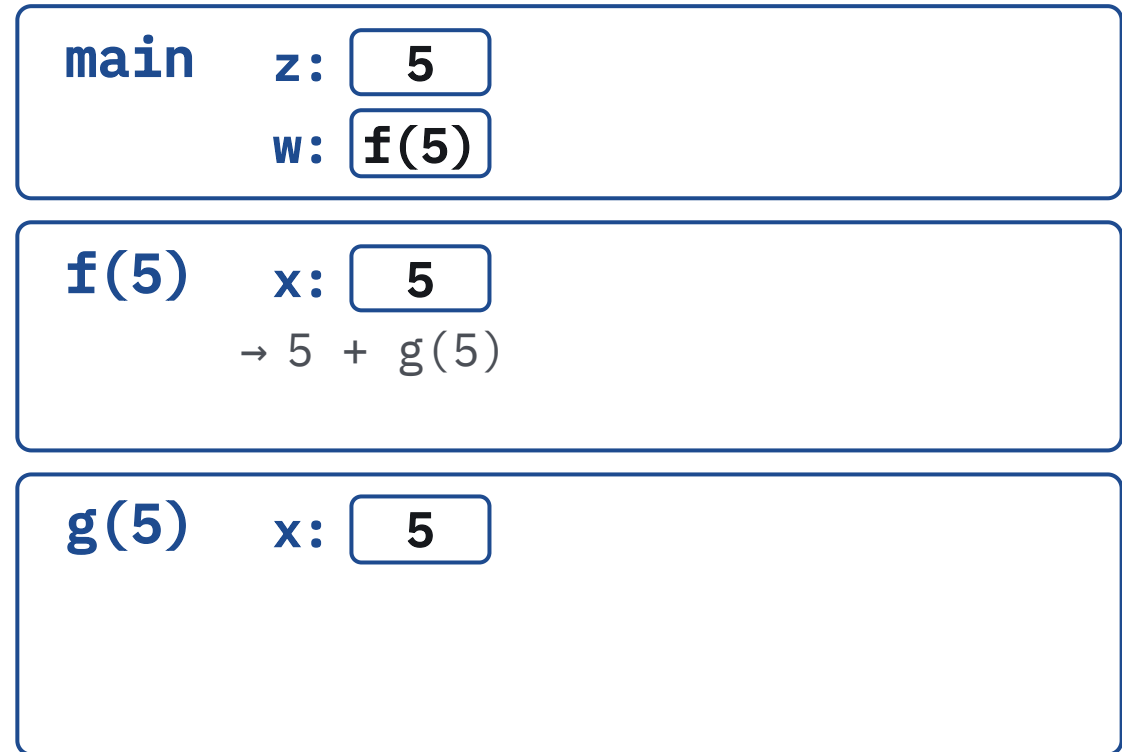
```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```



Calling `f(z)` pushes a new frame — a local scope for this call, with its own copy of the parameter `x`.

Stack diagrams

```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```



f(5) needs the value of `g(5)` before it can finish — another frame gets added.

Stack diagrams

```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```

main z: 5
 w: f(5)

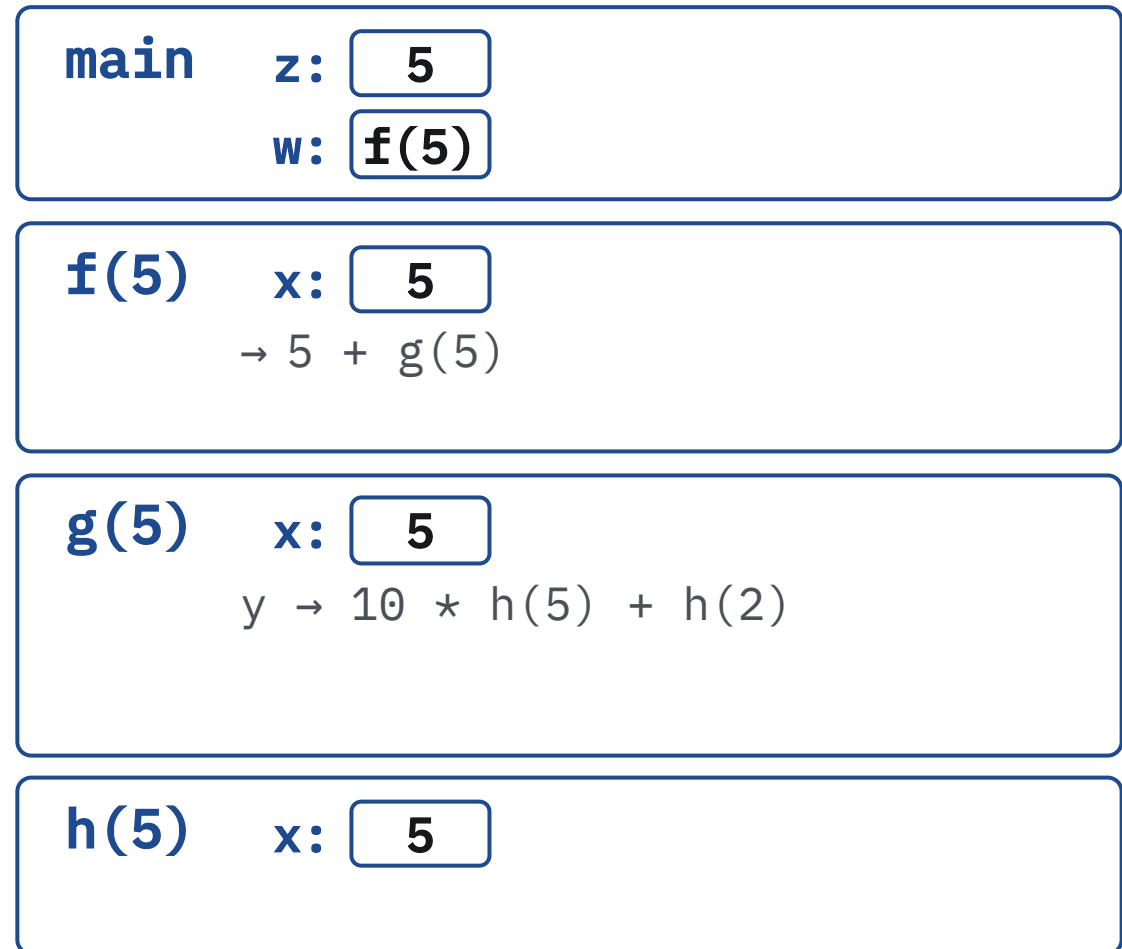
f(5) x: 5
 → 5 + g(5)

g(5) x: 5
 y → 10 * h(5) + h(2)

f(5) needs the value of g(5) before it can finish — another frame gets added.

Stack diagrams

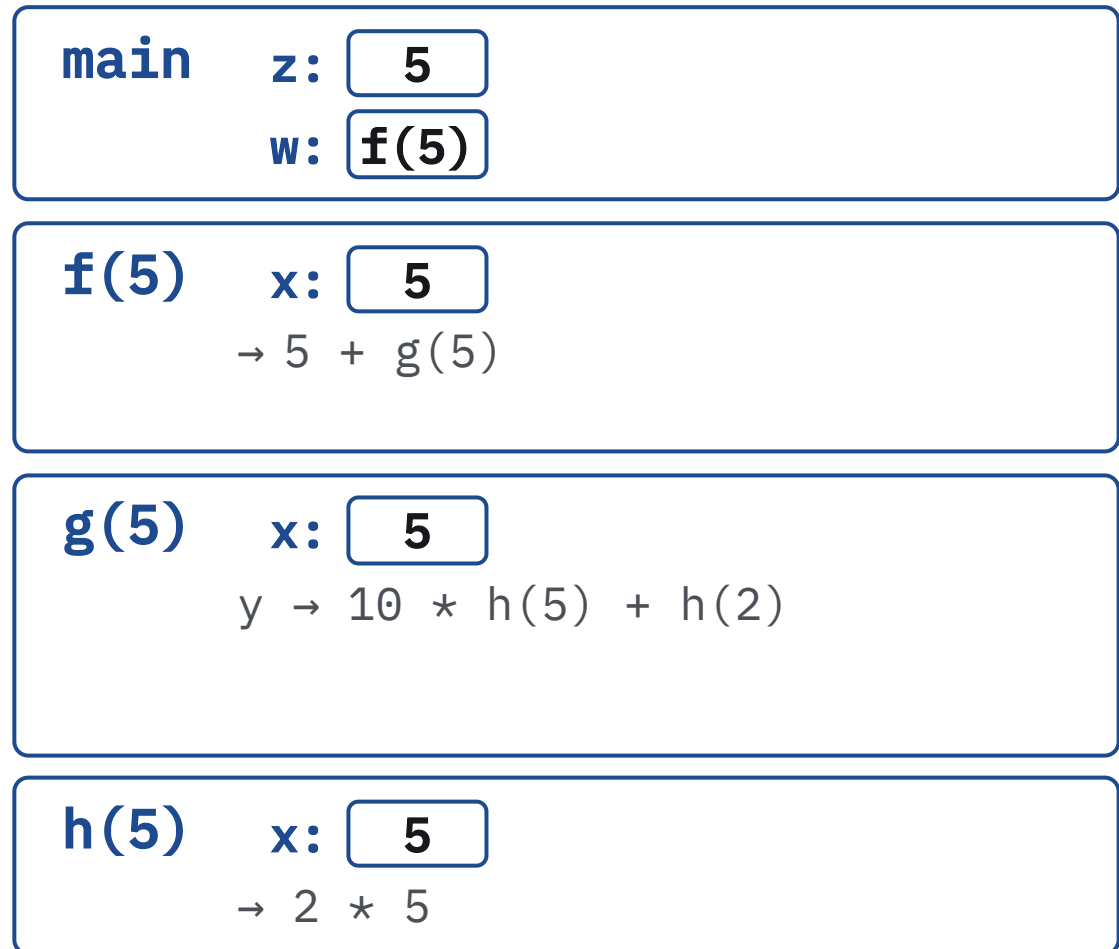
```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```



g(5) needs the value of `h(5)` before it can finish — another frame gets added.

Stack diagrams

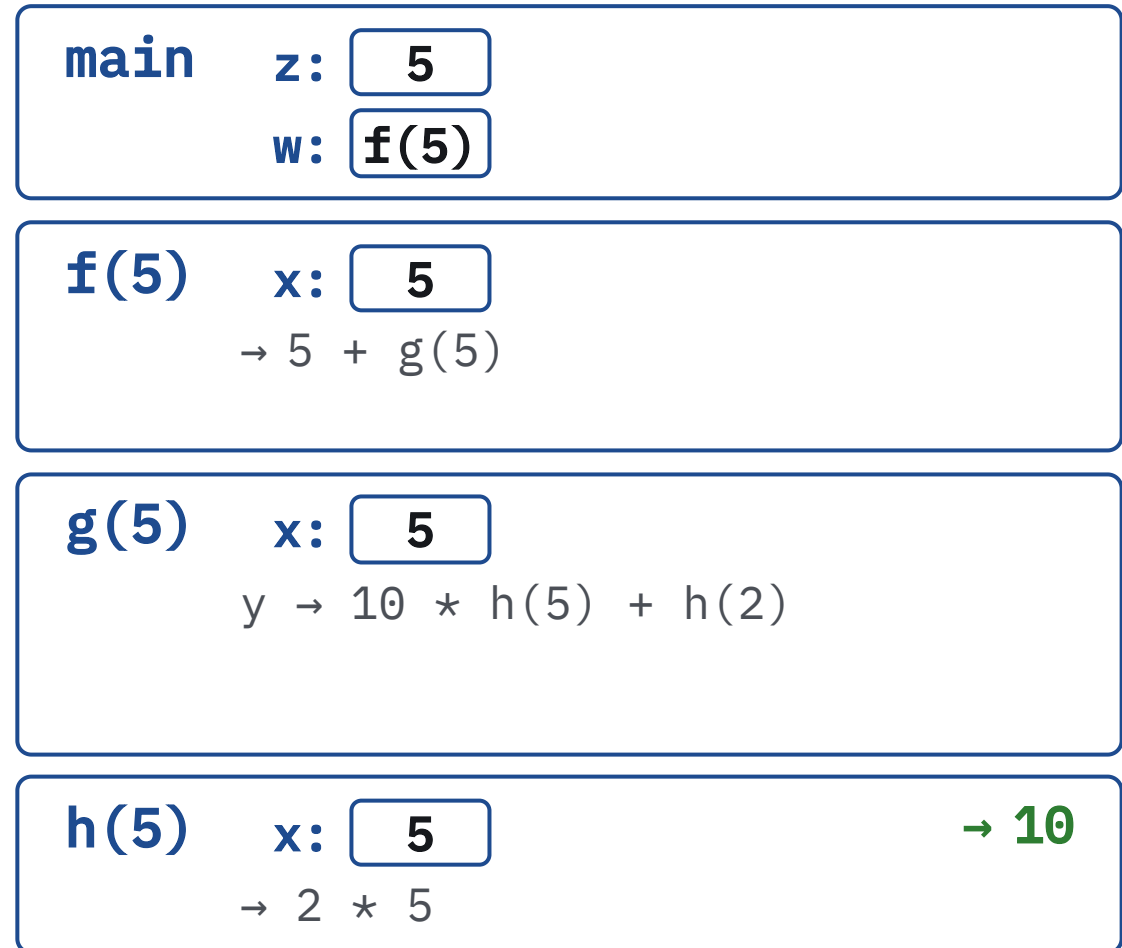
```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```



g(5) needs the value of `h(5)` before it can finish — another frame gets added.

Stack diagrams

```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```



h(5) is a one-line function — it pushes, resolves, and pops right away, handing 10 back to g.

Stack diagrams

```
def f(x):  
    return x + g(x)
```

```
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y
```

```
def h(x):  
    return 2 * x
```

```
z = 5  
w = f(z)
```

main **z:** 5
 w: f(5)

f(5) **x:** 5
 → 5 + g(5)

g(5) **x:** 5
 y → 10 * h(5) + h(2)
 y → 10 * 10 + h(2)

h(5) **x:** 5 → **10**

Stack diagrams

```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```

main z: 5
 w: f(5)

f(5) x: 5
 → 5 + g(5)

g(5) x: 5
 y → 10 * h(5) + h(2)
 y → 10 * 10 + h(2)

h(5) x: 5 → 10

h(2) x: 2

Stack diagrams

```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```

main **z:** 5
 w: f(5)

f(5) **x:** 5
 → 5 + g(5)

g(5) **x:** 5
 y → 10 * h(5) + h(2)
 y → 10 * 10 + h(2)

h(5) **x:** 5 → **10**

h(2) **x:** 2
 → 2 * 2

Stack diagrams

```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```

main **z:** 5
 w: f(5)

f(5) **x:** 5
 → 5 + g(5)

g(5) **x:** 5
 y → 10 * h(5) + h(2)
 y → 10 * 10 + h(2)

h(5) **x:** 5 → 10

h(2) **x:** 2 → 4
 → 2 * 2

Stack diagrams

```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```

main **z:** 5
 w: f(5)

f(5) **x:** 5
 → 5 + g(5)

g(5) **x:** 5 → **104**
 y → 10 * h(5) + h(2)
 y → 10 * 10 + h(2)
 y → 10 * 10 + 4 → 104

h(5) **x:** 5 → **10**

h(2) **x:** 2 → **4**

Once both h(5) and h(2) are known, g can finish computing y and return.

Stack diagrams

```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```

main z: 5
 w: f(5)

f(5) x: 5
 → 5 + g(5)
 → 5 + 104 → 109

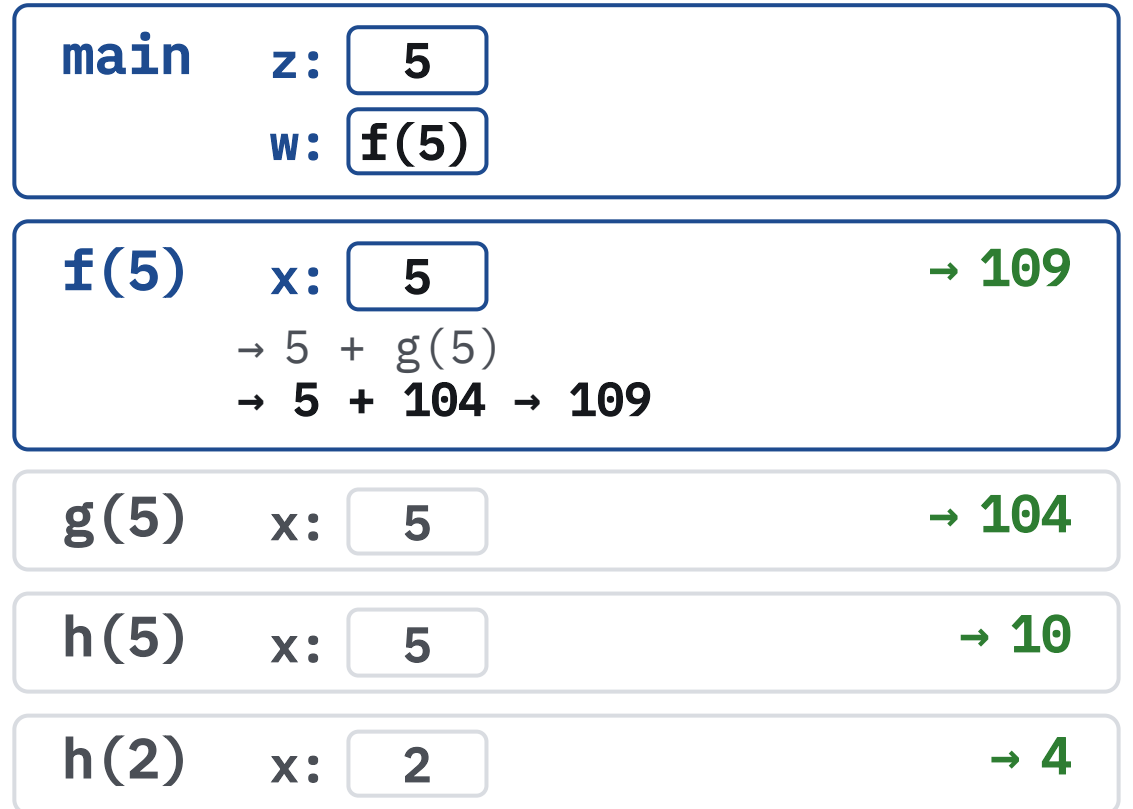
g(5) x: 5 → 104

h(5) x: 5 → 10

h(2) x: 2 → 4

Stack diagrams

```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```



Stack diagrams

```
def f(x):  
    return x + g(x)  
  
def g(x):  
    y = 10 * h(x) + h(x // 2)  
    return y  
  
def h(x):  
    return 2 * x  
  
z = 5  
w = f(z)
```



Once **f(z)** returns, its frame is popped too — we're back in **main**, with **w** holding the final answer, **109**.

Activity

1. Download [day03_stack.py](#) and save it in your `cs145/class` folder.
2. Launch Thonny. From the File menu, open `cs145/class/day03_stack.py`.
3. Draw the stack diagram on paper for the function call at the bottom. Create one frame per call, in the order they would be pushed to the call stack.
4. Run the code and check your trace against Python's actual output.

Worksheet: [day03-worksheet.pdf](#) · Answer key: [day03-worksheet-key.pdf](#)

To do

- Review this slide deck and these corresponding notes: [Notes 4](#)
- Continue working on [Homework 1](#), due this Thursday

Next time

Lecture 4 — modules and strings