

[220] Creating Functions and Understanding Function Scope

Department of Computer Sciences
University of Wisconsin-Madison

Readings:

Parts of Chapter 3 of Think Python,
Chapter 5.5 to 5.8 of Python for Everybody
Creating Fruitful Functions

Part I: Creating Functions

Learning Objectives

Explain the syntax of a function header:

- def, (), :, tabbing, return

Write a function with:

- correct header and indentation
- a return value (fruitful function) or without (void function)
- parameters that have default values

Write a function:

- knowing difference in outcomes of print and return statements

Determine result of function calls with 3 types of arguments:

- positional, keyword, and default

Trace function invocations, to determine control flow

Start with some Jupyter examples

pre-installed (e.g., math)

- sqrt()
- sin(), cos()
- pi, etc.

built in

- input()
- print()
- len()
- etc.

Where do **modules** come from?

installed (e.g., jupyter)

- pip install jupyter
- pip install ...

custom

- project (lab-p3)

Anaconda did these installations for us

Main Code:

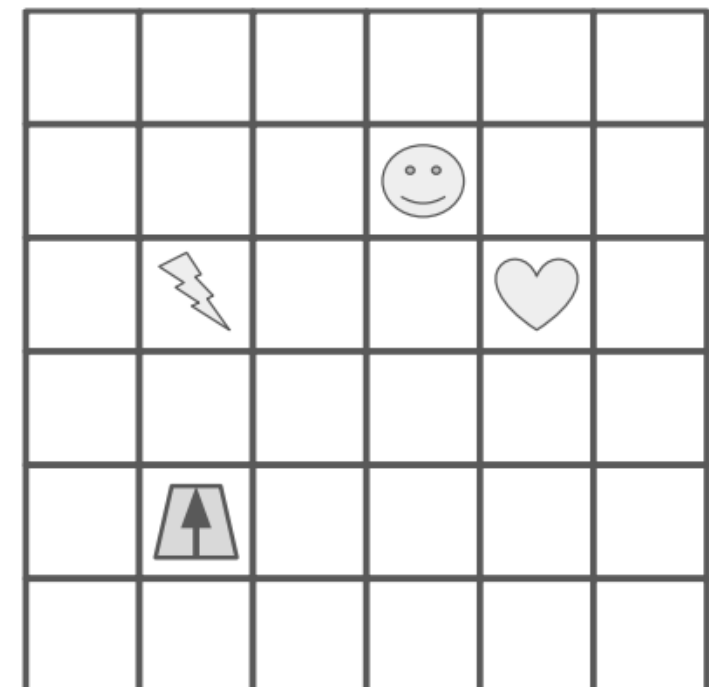
1. Put 2 in the “moves” box
2. Perform the steps under “Move Code”, then continue to step 3
3. Rotate the robot 90 degrees to the right (so arrow points to right)
4. Put 3 in the “moves” box
5. Perform the steps under “Move Code”, then continue to step 6
6. Whatever symbol the robot is sitting on, write that symbol in the “resut” box

Move Code:

- A. If “moves” is 0, stop performing these steps in “Move Code”, and go back to where you last were in “Main Code” to complete more steps
- B. Move the robot forward one square, in the direction the arrow is pointing
- C. Decrease the value in “moves” by one
- D. Go back to step A

*how do we write functions
like move code?*

Functions are like “mini programs”,
as in our robot worksheet problem



Types of functions

Sometimes functions **do** things

- Like “Move Code”
- May produce output with print
- May change variables

Sometimes functions **produce** values

- Similar to mathematical functions
- Many might say a function “**returns a value**”
- Downey calls these functions “**fruitful**” functions
(we’ll use this, but don’t expect people to generally be aware of this terminology)

Sometimes functions do both!

Types of functions

Sometimes functions **do** things

- Like “Move Code”
- May produce output with print
- May change variables

Sometimes functions **produce** values

- Similar to mathematical functions
- Many might say a function “**returns a value**”
- Downey calls these functions “**fruitful**” functions
(we’ll use this, but don’t expect people to generally be aware of this terminology)

Sometimes functions do both!

Math to Python

Math: $f(x) = x^2$

Python:

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

Math to Python

Math:

$$f(x) = x^2$$

Python:

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

Function name is “f”

Math to Python

Math:

$$f(x) = x^2$$

Python:

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

It takes one parameter, "x"

Math to Python

Math:

$$f(x) = x^2$$

Python:

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



In Python, start a function definition with “def” (short for definition), and use a colon (“:”) instead of an equal sign (“=”)

Math to Python

Math:

$$f(x) = x^2$$

Python:

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

2

In Python, put the “return” keyword before the expression associated with the function

Math to Python

Math: $f(x) = x^2$

Python:

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

3

In Python, indent (tab space) before the statement(s)

Math to Python

Math: $g(r) = \pi r^2$

Python:

```
def g(r):  
    return 3.14 * r ** 2
```

Math to Python

Math: $g(r) = \pi r^2$

Python:

```
def get_area(radius):  
    return 3.14 * radius ** 2
```


Math to Python

Math: $g(r) = \pi r^2$

Python:

```
def get_area(diameter):  
    radius = diameter / 2  
    return 3.14 * radius ** 2
```

Let's implement functions

```
cube(side)
```

```
is_between(lower, num, upper)
```

jupyter / PythonTutor demos ...

Rules for filling parameters...

function declaration

```
def foo(x, y=-1):
```

x =

???

y =

???

not actual code, but imagine
parameters as variables that are
automatically initialized for you

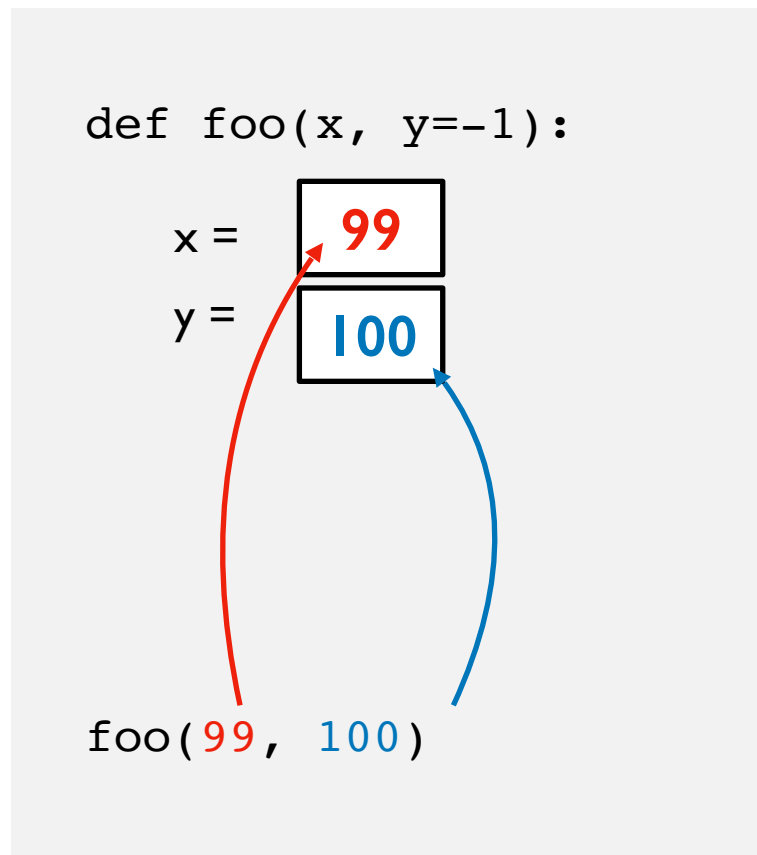
function invocation

```
foo(99, 100)
```



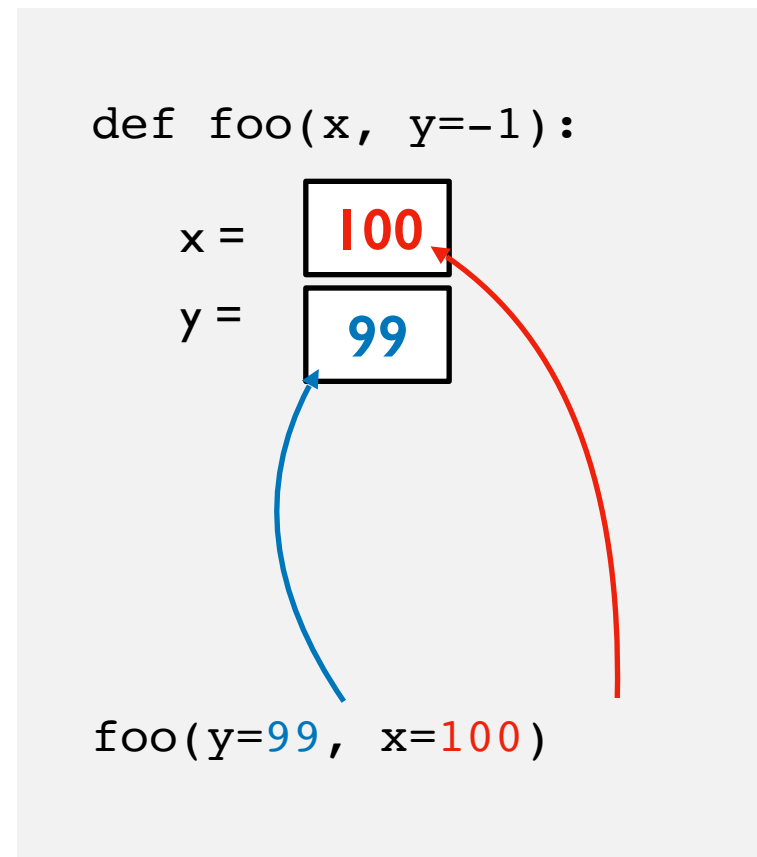
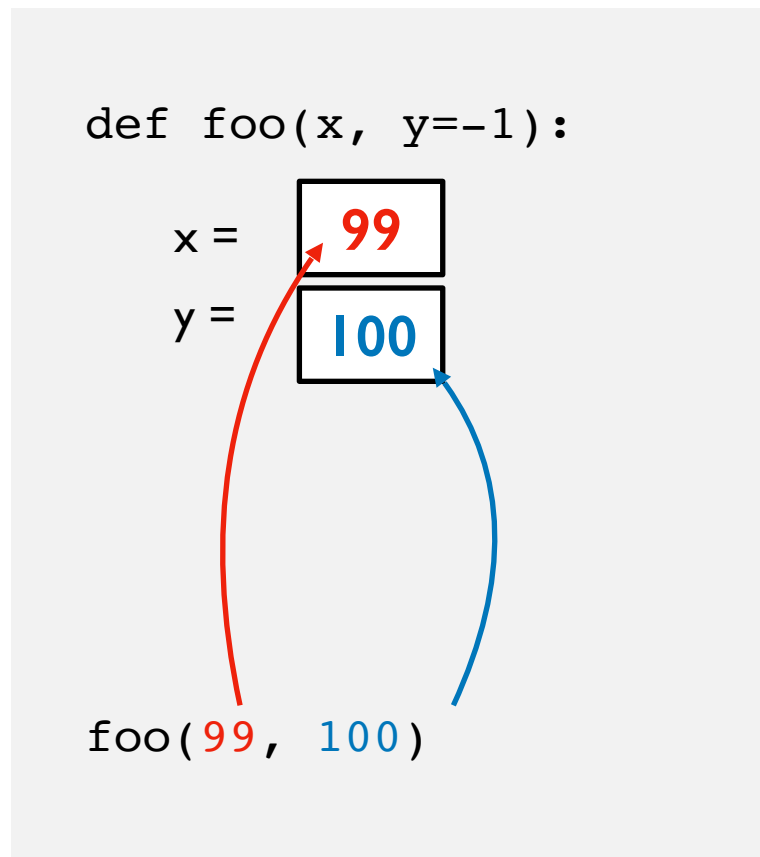
positional arguments

Rules for filling parameters...



positional arguments

Rules for filling parameters...



1

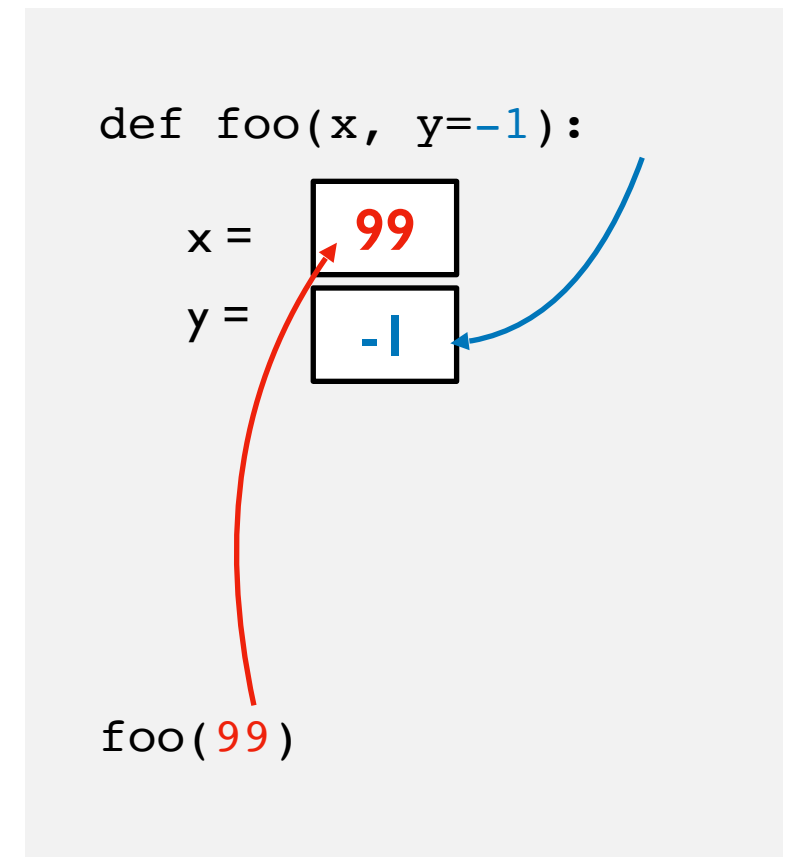
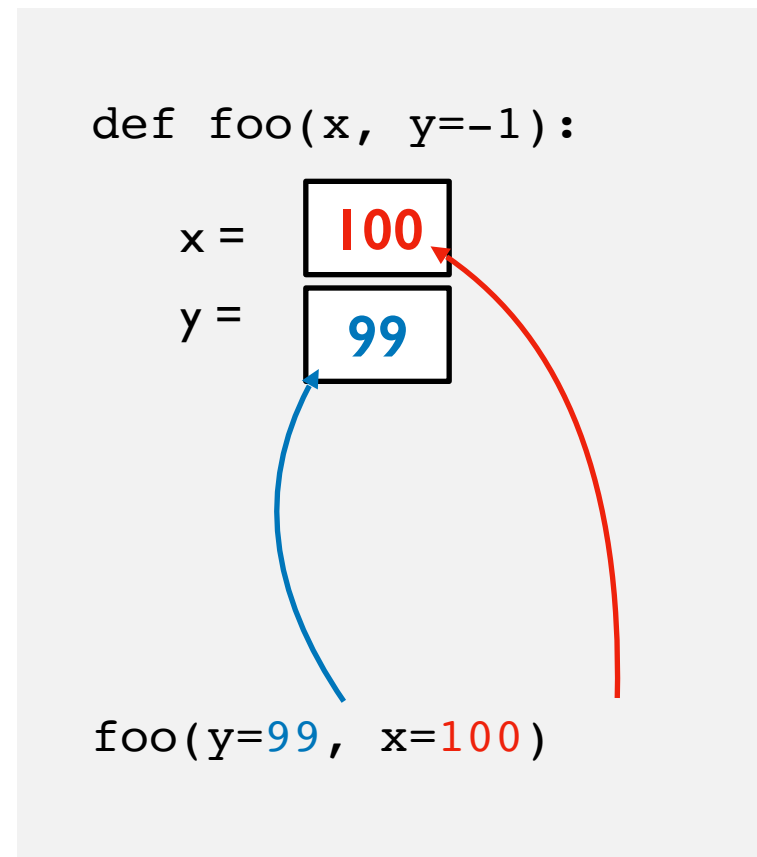
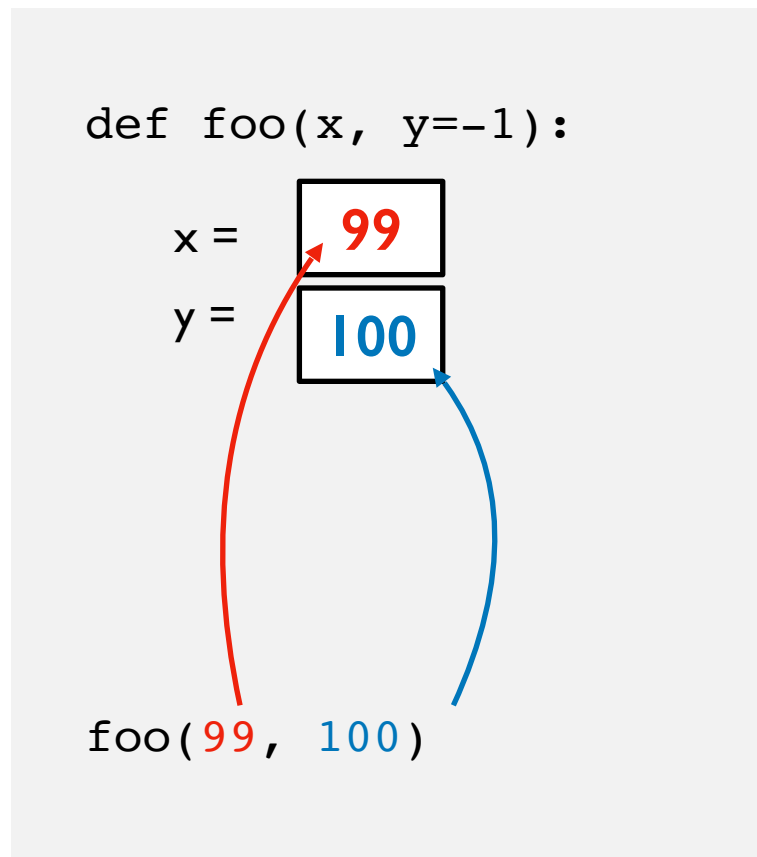
positional arguments

2

keyword arguments



Rules for filling parameters...



1

positional arguments

2

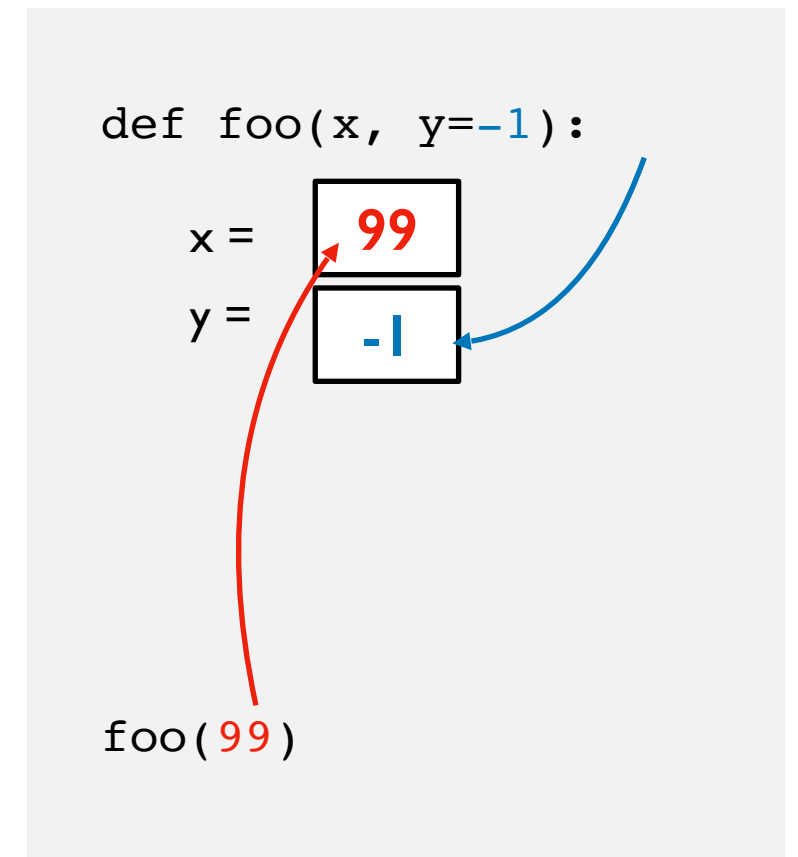
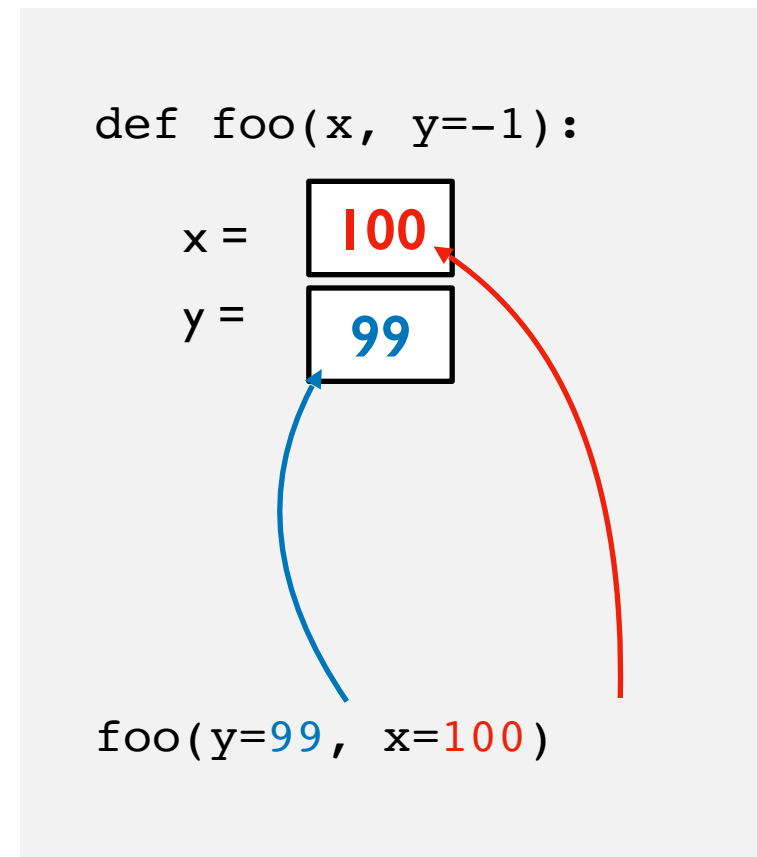
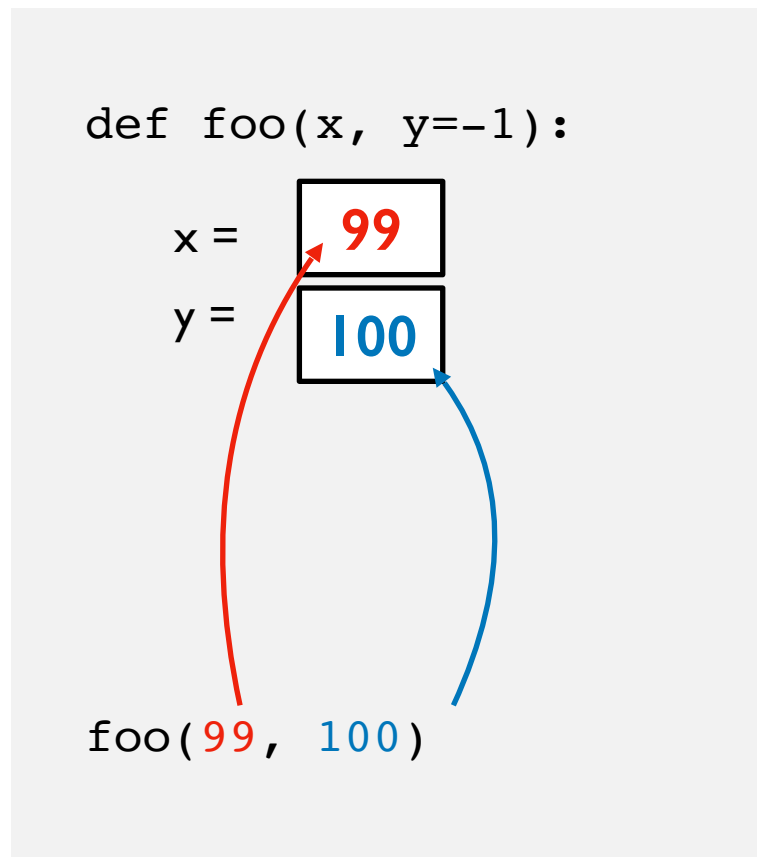
keyword arguments

3

default arguments

common pitfall: confusing keyword arguments and default arguments

Rules for filling parameters...



1

positional arguments

2

keyword arguments

3

default arguments

worksheet practice...

Generating grid for game like Battleship

Grid:

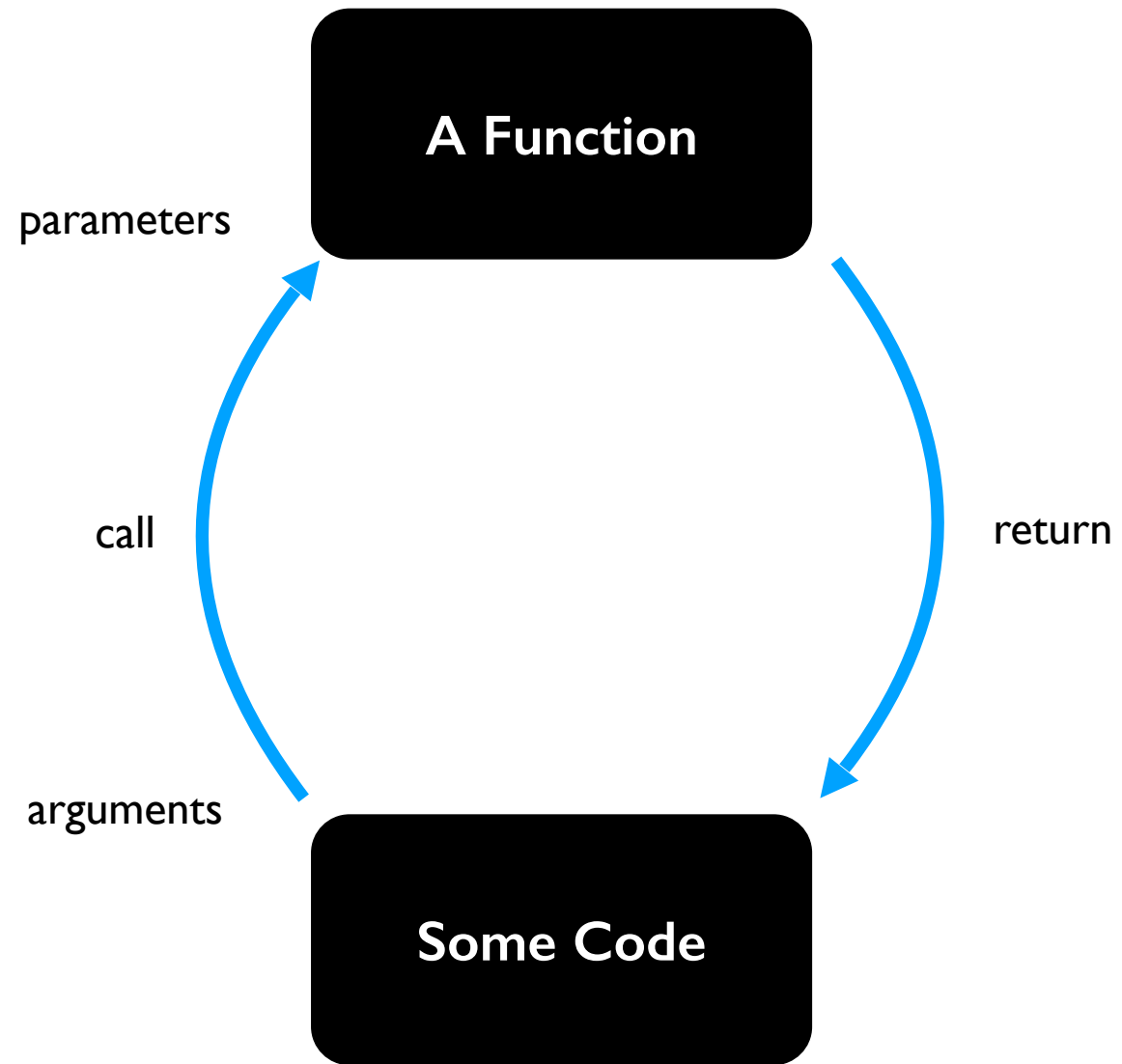
```
#####  
#####  
#####  
#####  
#####  
#####  
#####  
#####  
#####
```

10 x 8 grid

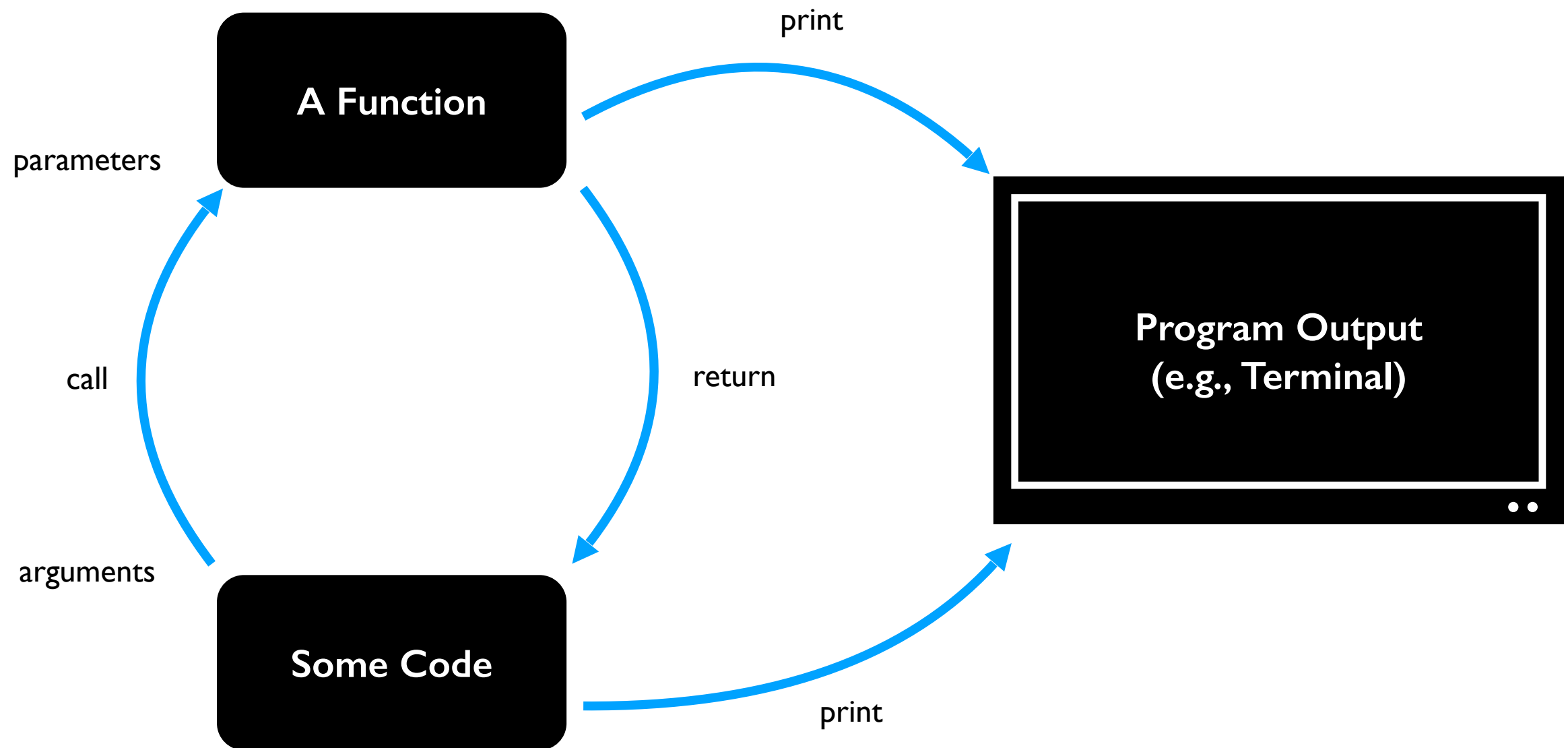
```
get_grid(width, height, symb = '#', title = 'Grid:')
```

Jupyter / PythonTutor demo...

Print vs. Return

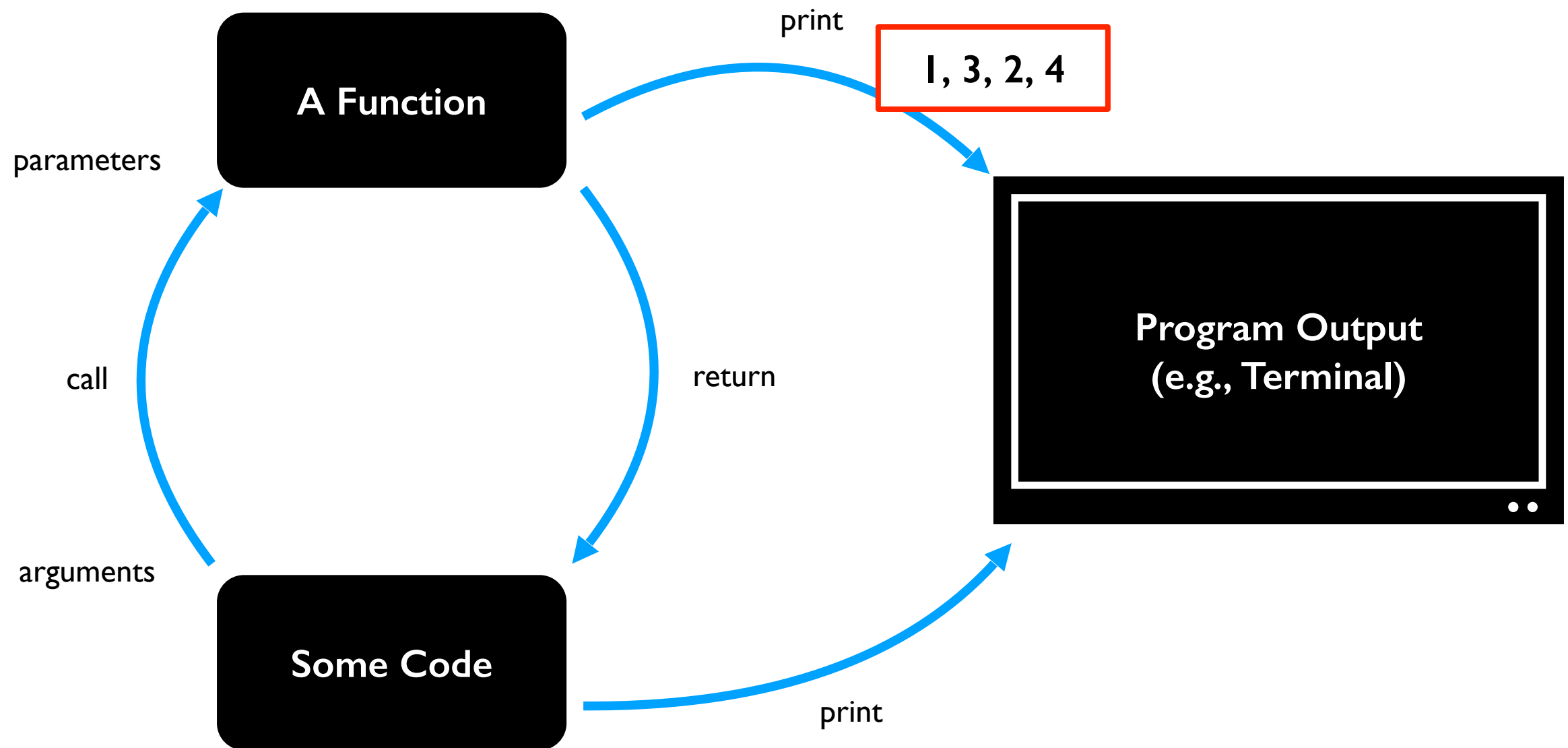


Print vs. Return

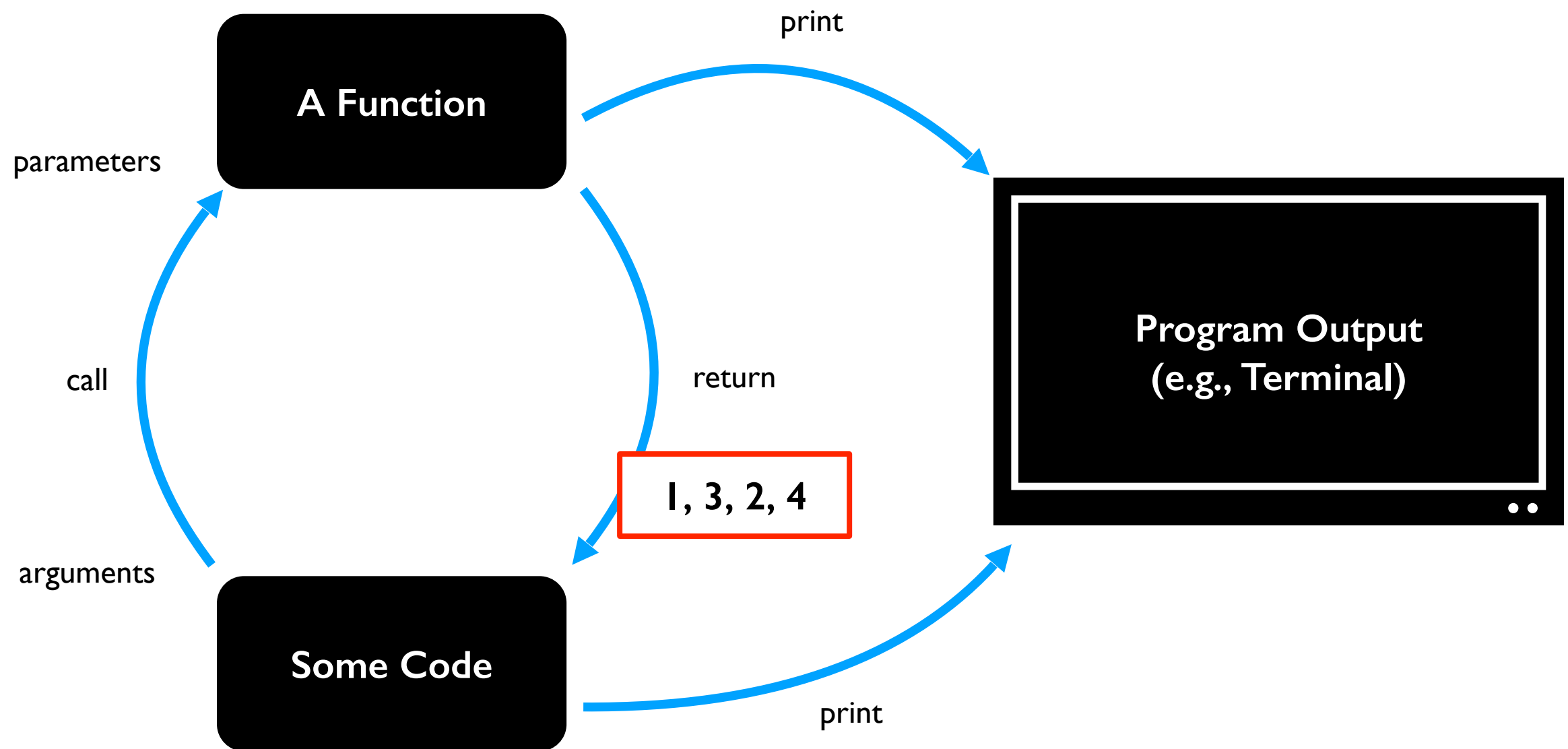


we could call print from multiple places

Print vs. Return

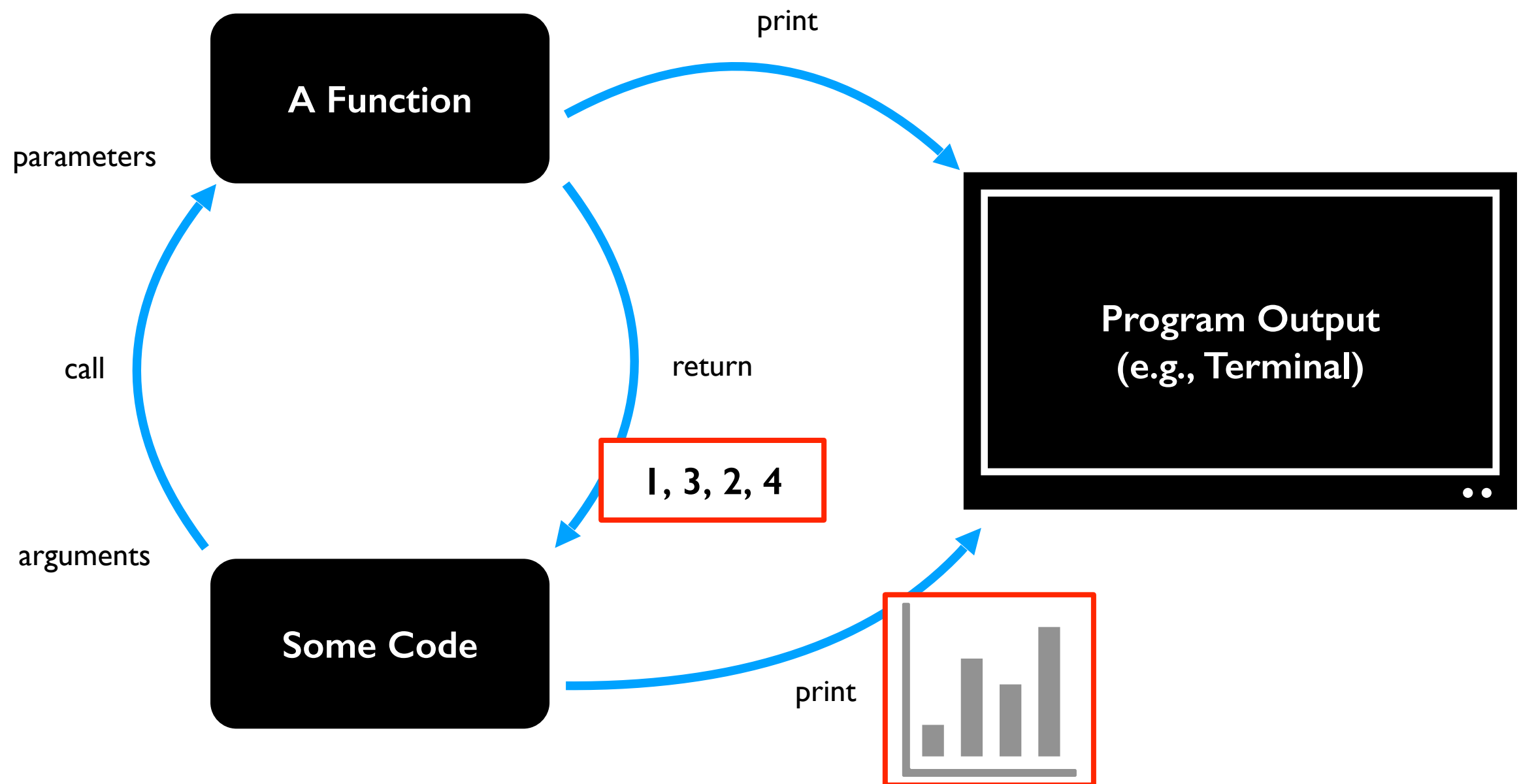


Print vs. Return



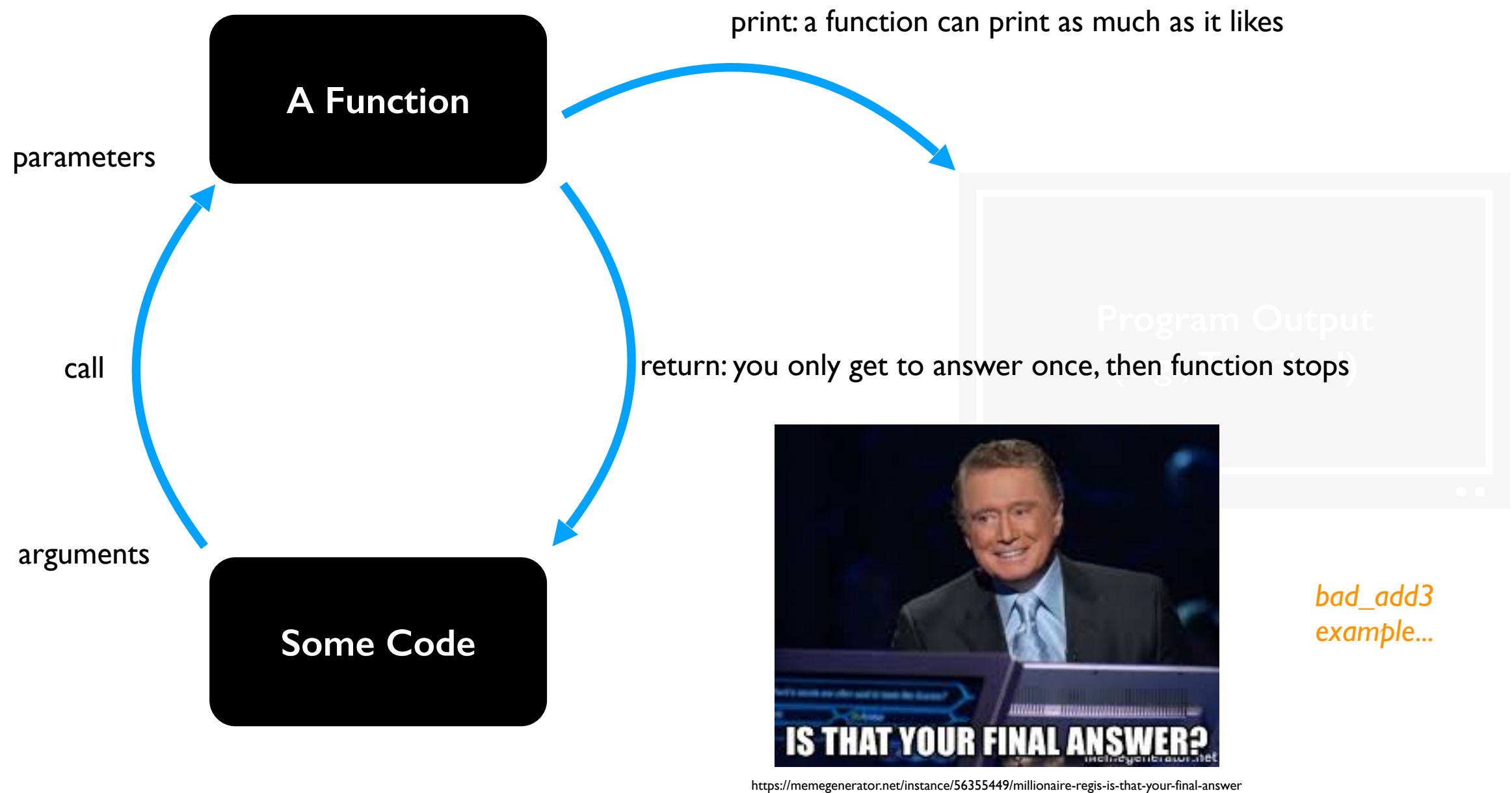
returning, instead of printing, gives callers different options for how to use the result

Print vs. Return



returning, instead of printing, gives callers different options for how to use the result

Print vs. Return



returning, instead of printing, gives callers different options for how to use the result

Interactive Examples with PythonTutor

```
def func_c():  
    print("C")  
  
def func_b():  
    print("B1")  
    func_c()  
    print("B2")  
  
def func_a():  
    print("A1")  
    func_b()  
    print("A2")  
  
func_a()
```

*Let's trace this example.
(Problem 21 from the
worksheet)*

Part 2: Understanding Function Scope

Learning Objectives

Explain rules of scope of **local variables** in a function

- When are they created?
- When are they destroyed?
- What parts of a program have access to them? (frames)

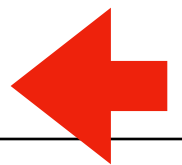
Understand **global variables**

- How can they be used and modified within a function?(global keyword)
- Where are they stored? (global frame)
- What parts of a program have access to them?
- How can they be mis-represented as local variables?

Read: Downey Ch 3 ("Parameters and Arguments" to end)

[Link to Slides](#)

[Interactive Exercises](#)



Explain **argument passing**

- “pass by value”

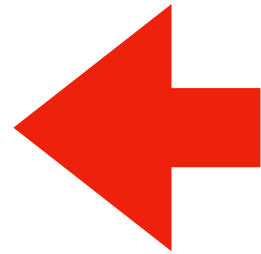
don't memorize the examples,
learn the rules of Python

sample question: why did Python
do this thing I didn't expect
at this specific line (ask us!)

Today's Outline

Context

- Examples



Frames

Demos: Local Variables

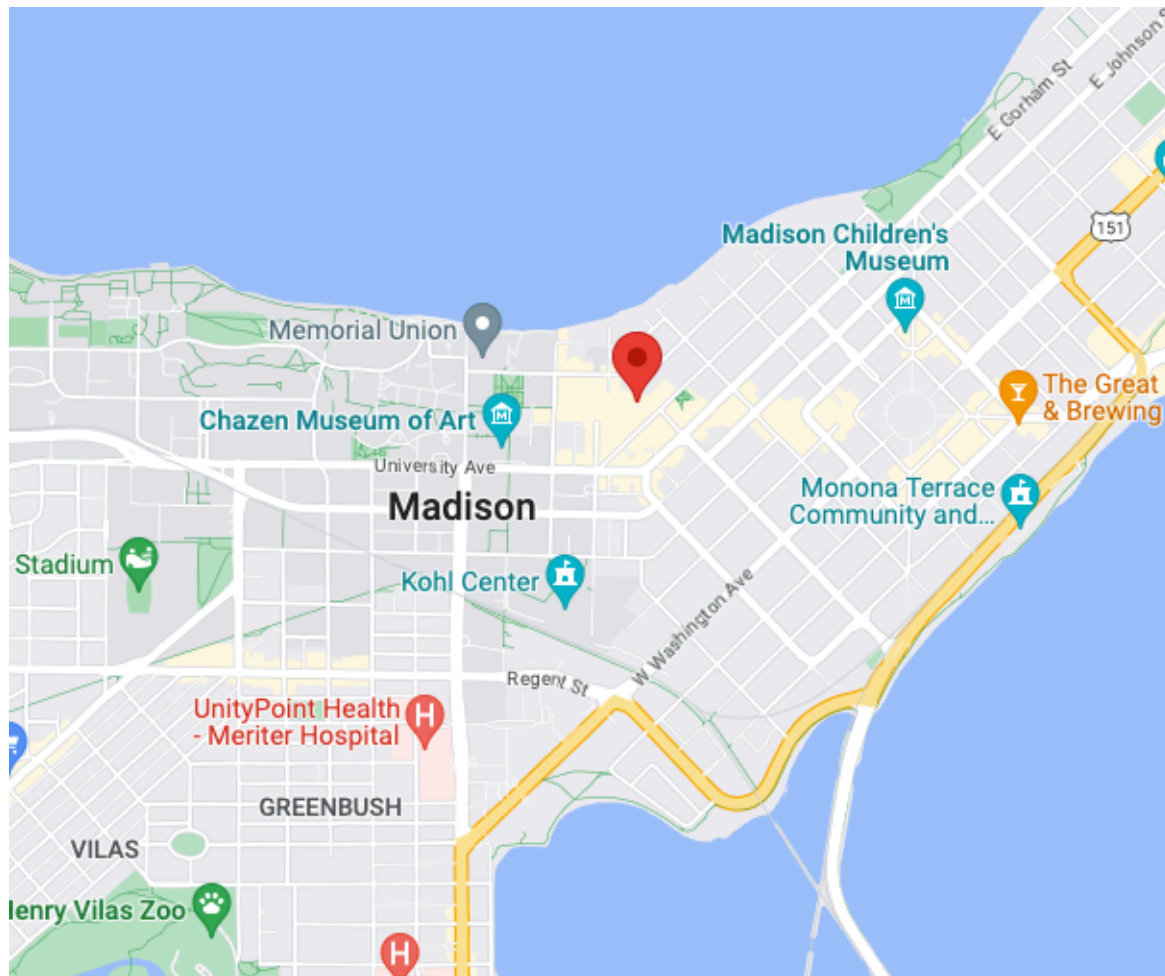
Demos: Global Variables

Demos: Argument Passing

Context

Often (in life and programming), the same name can mean different things in different contexts

- Examples?
- Human name: **Anna** (are we talking about CS220 or Frozen?)
- Street address: **534 State Street** (what city are we in?)
- Files: **test.py** (which directory are we in?)



Context

Often (in life and programming), the same name can mean different things in different contexts

- Examples?
- Human name: **Anna** (are we talking about CS220 or Frozen?)
- Street address: **534 State Street** (what city are we in?)
- Files: **test.py** (which directory are we in?)

Our code often have different variables with the same name

- How do we keep variable names organized?
- How do we know what a variable name is referring to?

Context

Often (in life and programming), the same name can mean different things in different contexts

- Examples?
- Human name: **Anna** (are we talking about CS220 or Frozen?)
- Street address: **534 State Street** (what city are we in?)
- Files: **test.py** (which directory are we in?)

Our code often have different variables with the same name

- How do we keep variable names organized?
- How do we know what a variable name is referring to?

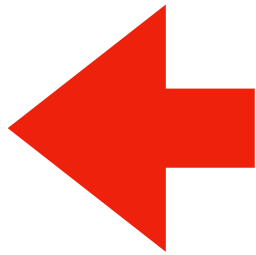
with groups called “frames”

we’ll learn some
rules for this

Today's Outline

Context

Frames



Demos: Local Variables

Demos: Global Variables

Demos: Argument Passing

Frames

Every time a function is invoked (i.e., called), the invocation gets a new “**frame**” for holding variables

- The parameters also exist in a frame

Global frame

- There is always one global frame that all functions can access

When a variable name is used, Python looks two places:

- 1 the function invocation's frame
- 2 the global frame

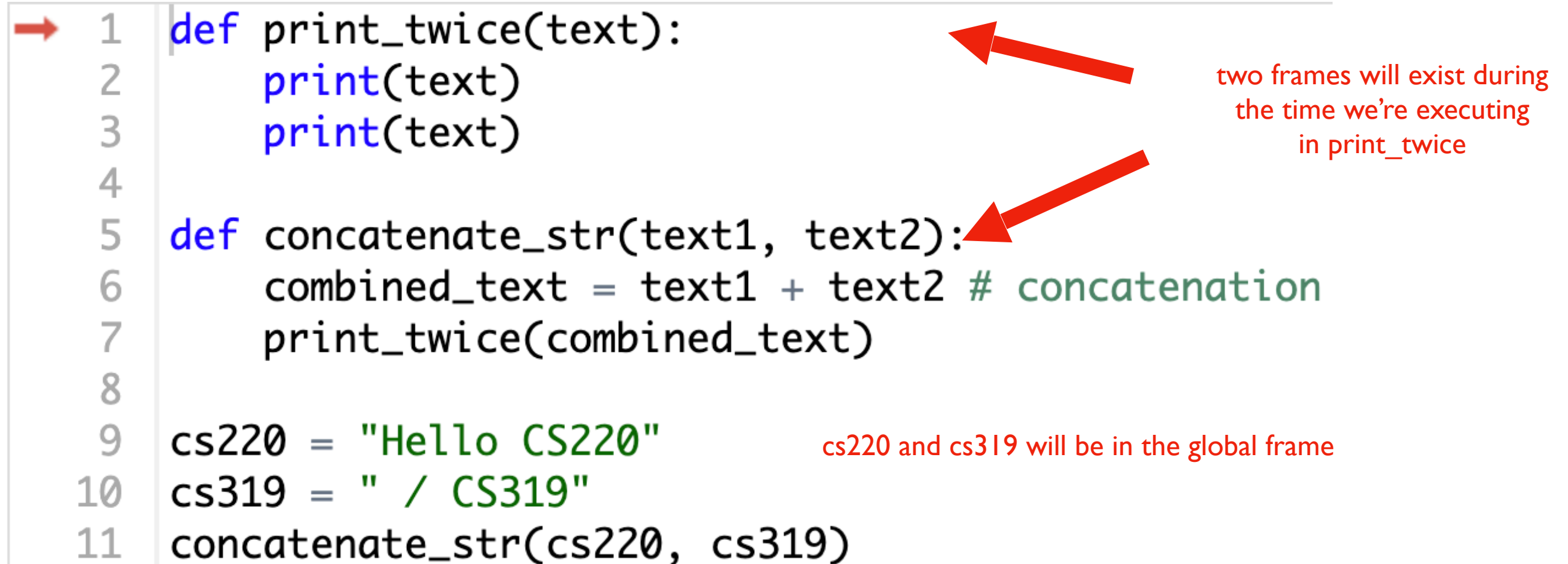
Understanding scope: example

```
→ 1 def print_twice(text):  
2     print(text)  
3     print(text)  
4  
5 def concatenate_str(text1, text2):  
6     combined_text = text1 + text2 # concatenation  
7     print_twice(combined_text)  
8  
9 cs220 = "Hello CS220"  
10 cs319 = " / CS319"  
11 concatenate_str(cs220, cs319)
```


Understanding scope: example

```
→ 1 def print_twice(text):  
  2     print(text)  
  3     print(text)  
  4  
  5 def concatenate_str(text1, text2):  
  6     combined_text = text1 + text2 # concatenation  
  7     print_twice(combined_text)  
  8  
  9 cs220 = "Hello CS220" cs220 and cs319 will be in the global frame  
 10 cs319 = " / CS319"  
 11 concatenate_str(cs220, cs319)
```

Understanding scope: example



```
→ 1 def print_twice(text):  
2     print(text)  
3     print(text)  
4  
5 def concatenate_str(text1, text2):  
6     combined_text = text1 + text2 # concatenation  
7     print_twice(combined_text)  
8  
9 cs220 = "Hello CS220"  
10 cs319 = " / CS319"  
11 concatenate_str(cs220, cs319)
```

two frames will exist during the time we're executing in print_twice

cs220 and cs319 will be in the global frame

Understanding scope: example

```
→ 1 def print_twice(text):  
2     print(text)  
3     print(text)  
4  
5 def concatenate_str(text1, text2):  
6     combined_text = text1 + text2 # concatenation  
7     print_twice(combined_text)  
8  
9 cs220 = "Hello CS220"  
10 cs319 = " / CS319"  
11 concatenate_str(cs220, cs319)
```

two frames will exist during the time we're executing in print_twice

cs220 and cs319 will be in the global frame

you don't generally see or interact with frames when programming, but it's an important mental model

Understanding scope: example

```
→ 1 def print_twice(text):  
2     print(text)           can access: cs220, cs319, text  
3     print(text)  
4  
5 def concatenate_str(text1, text2): can access: cs220, cs319, text1, text2,  
6     combined_text = text1 + text2 combined_text  
7     print_twice(combined_text) # concatenation  
8  
9 cs220 = "Hello CS220"  
10 cs319 = " / CS319"  
11 concatenate_str(cs220, cs319) this code can access: line1, line2
```

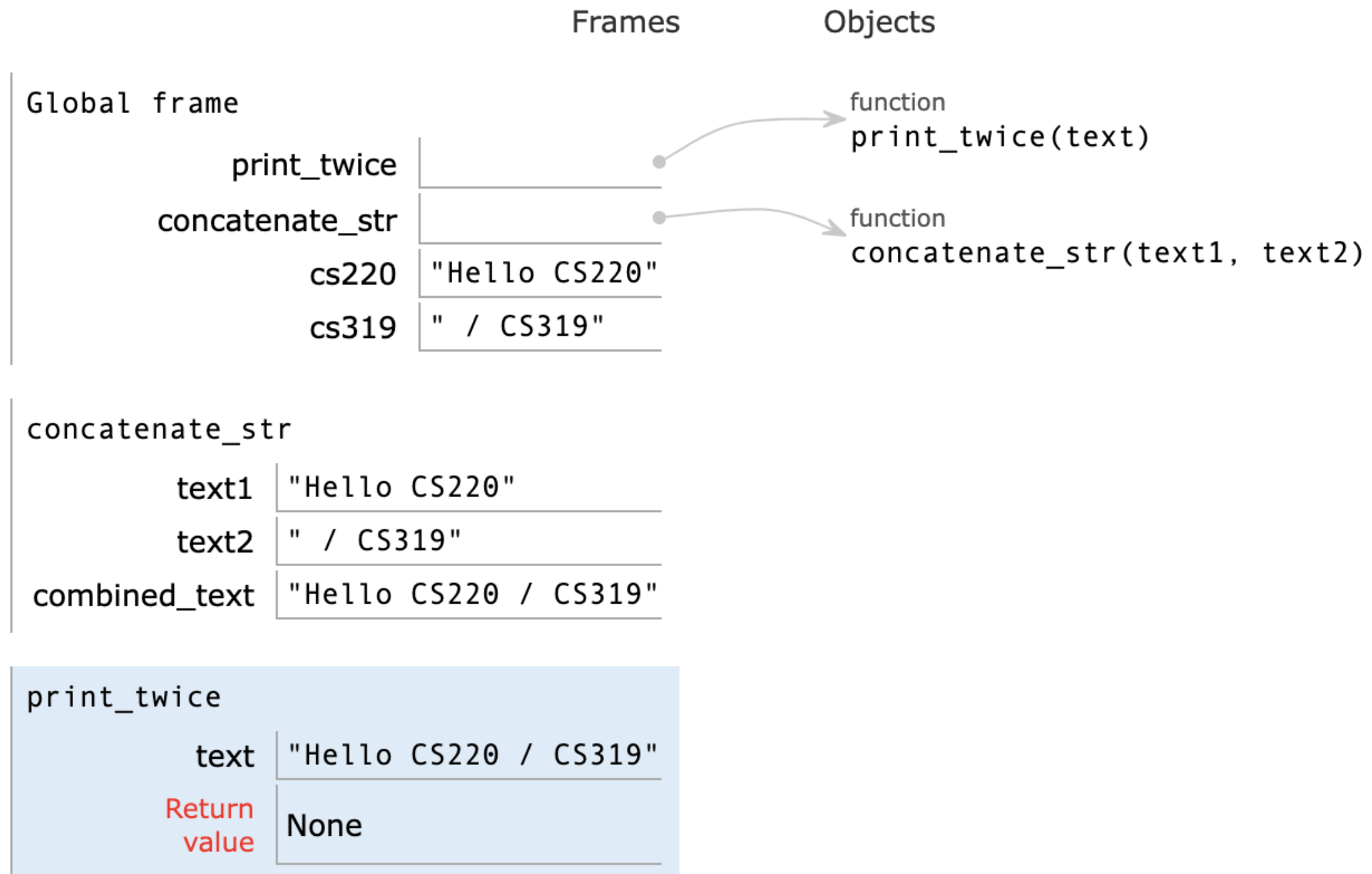
we call the variables that can currently be
accessed “in scope” and variables that
cannot be “out of scope”

Understanding scope: example

```
→ 1 def print_twice(text):  
2     print(text)  
3     print(text)  
4  
5 def concatenate_str(text1, text2):  
6     combined_text = text1 + text2 # concatenation  
7     print_twice(combined_text)  
8  
9 cs220 = "Hello CS220"  
10 cs319 = " / CS319"  
11 concatenate_str(cs220, cs319)
```

Arguments are copied to parameters:
this is called "pass by value"

Understanding scope: example (PythonTutor)



Understanding scope: example

```
def print_twice(text):  
    print(text)  
    print(text)  
  
def concatenate_str(text1, text2):  
    combined_text = text1 + text2 #  
concatenation  
    print_twice(combined_text)  
  
cs220 = "Hello CS220"  
cs319 = " / CS319"  
concatenate_str(cs220, cs319)
```

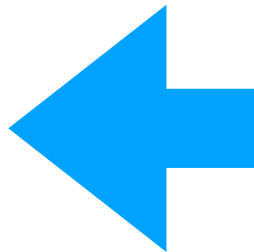
Try this example yourself using PythonTutor

Today's Outline

Context

Frames

Demos: Local Variables



Demos: Global Variables

Demos: Argument Passing

Let's do some examples in PythonTutor

Lessons about Local Variables

```
def set_x():  
    x = 100
```

```
print(x)
```

Lesson 1: functions don't execute unless they're called

Lessons about Local Variables

```
def set_x():  
    x = 100
```

```
set_x()  
print(x)
```

Lesson 2: variables created in a function die after function returns

Lessons about Local Variables

```
def count():  
    x = 1  
    x += 1  
    print(x)
```

```
count()  
count()  
count()
```

Lesson 3: variables start fresh every time a function is called again

Lessons about Local Variables

```
def display_x():  
    print(x)
```

```
def main():  
    x = 100  
    display_x()
```

```
main()
```

Lesson 4: you can't see the variables of other function invocations, even those that call you

Today's Outline

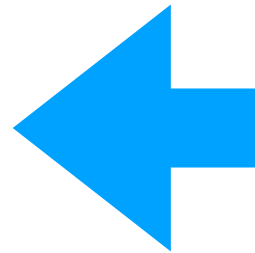
Context

Frames

Demos: Local Variables

Demos: Global Variables

Demos: Argument Passing



Lessons about Global Variables

```
msg = 'hello' # global, outside any func
```

```
def greeting():  
    print(msg)
```

```
print('before: ' + msg)  
greeting()  
print('after: ' + msg)
```

Lesson 5: you can generally just use global variables inside a function

Lessons about Global Variables

```
msg = 'hello'
```

```
def greeting():  
    msg = 'welcome!'  
    print('greeting: ' + msg)
```

```
print('before: ' + msg)  
greeting()  
print('after: ' + msg)
```

Lesson 6: if you do an assignment to a variable in a function, Python assumes you want it local

Lessons about Global Variables

```
msg = 'hello'
```

```
def greeting():  
    print('greeting: ' + msg)  
    msg = 'welcome!'
```

```
print('before: ' + msg)  
greeting()  
print('after: ' + msg)
```

Lesson 7: assignment to a variable should be before its use in a function, even if there's a global variable with the same name

Lessons about Global Variables

```
msg = 'hello'
```

```
def greeting():  
    global msg  
    print('greeting: ' + msg)  
    msg = 'welcome!'
```

```
print('before: ' + msg)  
greeting()  
print('after: ' + msg)
```

Lesson 8: use a global declaration to prevent Python from creating a local variable when you want a global variable

Today's Outline

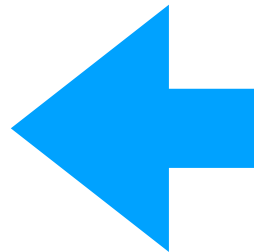
Context

Frames

Demos: Local Variables

Demos: Global Variables

Demos: Argument Passing



Lessons about Argument Passing

```
def f(x):  
    x = 'B'  
    print('inside: ' + x)
```

```
val = 'A'  
print('before: ' + val)  
f(val)  
print('after: ' + val)
```

Lesson 9: in Python, arguments are "passed by value", meaning
reassignments to a parameter don't change the argument outside

Lessons about Argument Passing

```
x = 'A'
```

```
def f(x):  
    x = 'B'  
    print('inside: ' + x)
```

```
print('before: ' + x)  
f(x)  
print('after: ' + x)
```

Lesson 10: it's irrelevant whether the argument (outside) and parameter (inside) have the same variable name

Lesson Summary

Local

Lesson 1: functions don't execute unless they're called

Lesson 2: variables created in a function die after function returns

Lesson 3: variables start fresh every time a function is called again

Lesson 4: you can't see the variables of other function invocations, even those that call you

Global

Lesson 5: you can generally just use global variables inside a function

Lesson 6: if you do an assignment to a variable in a function, Python assumes you want it local

Lesson 7: assignment to a variable should be before its use in a function, even if there's a global variable with the same name

Lesson 8: use a global declaration to prevent Python from creating a local variable when you want a global variable

Parameters

Lesson 9: in Python, arguments are "passed by value", meaning reassignments to a parameter don't change the argument outside

Lesson 10: it's irrelevant whether the argument (outside) and parameter (inside) have the same variable name