http://www.cs.cornell.edu/courses/cs1110/2018sp

Lecture 4: Defining Functions

(Ch. 3.4-3.11)

CS 1110

Introduction to Computing Using Python

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]

Things to Do Before Next Class

Readings:

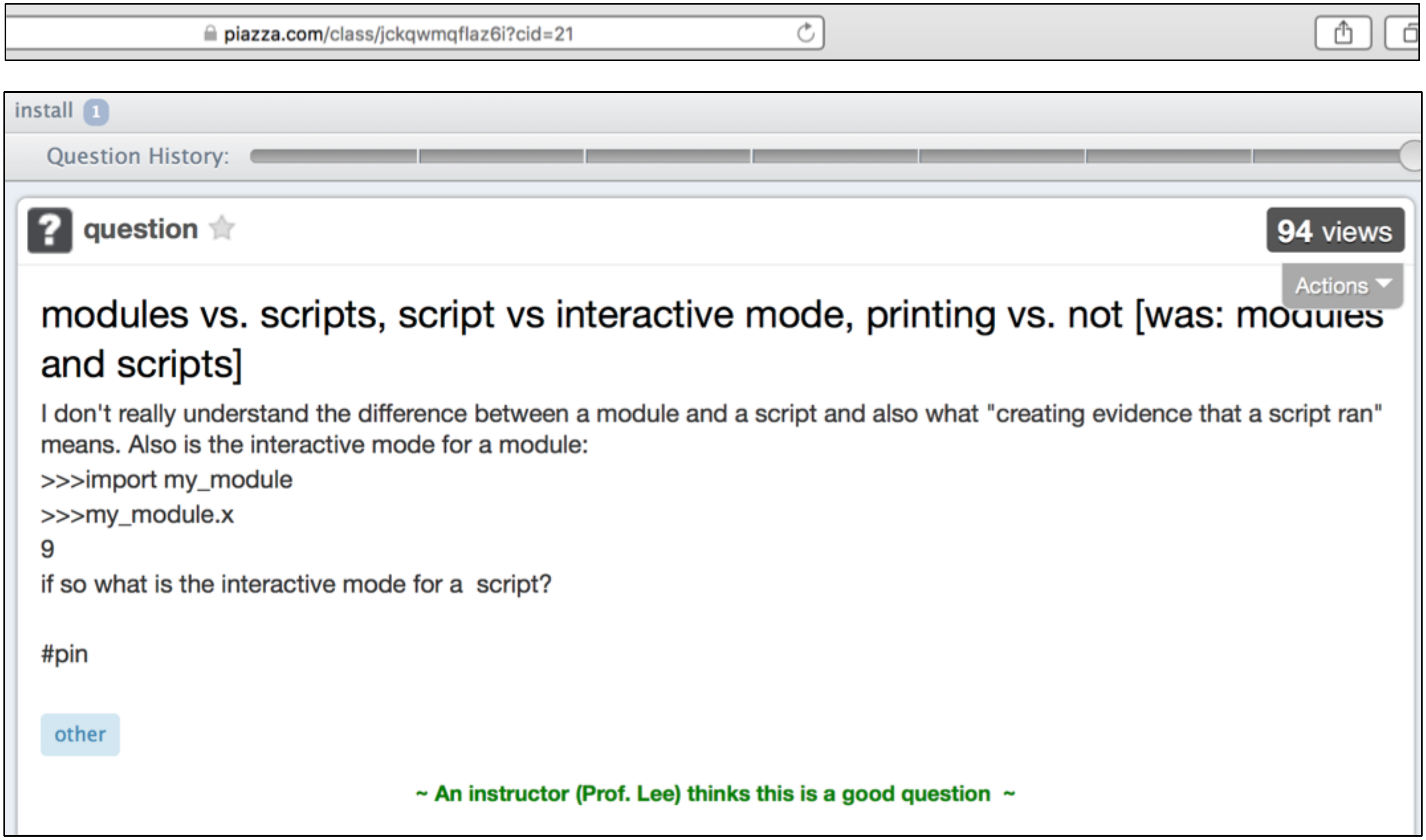
- Sections 8.1, 8.2, 8.4, 8.5, first paragraph of 8.9

Labs:

- Go to Lab! (Lab 2 is this week)
- Get Credit for Lab 1:
  - **can** be checked off during Tuesday's consulting hours 4:30-9:30 in the ACCEL lab
  - **cannot** be checked off after 3:45pm Wednesday
  - **check** online if you received credit:

http://www.cs.cornell.edu/courses/cs1110/2018sp/labs/index.php

Check out Piazza Post @21



piazza.com/class/jckqwmqflaz6i?cid=21

install 1

Question History:

question

94 views

Actions

modules vs. scripts, script vs interactive mode, printing vs. not [was: modules and scripts]

I don't really understand the difference between a module and a script and also what "creating evidence that a script ran" means. Also is the interactive mode for a module:

```
>>>import my_module
>>>my_module.x
9
```

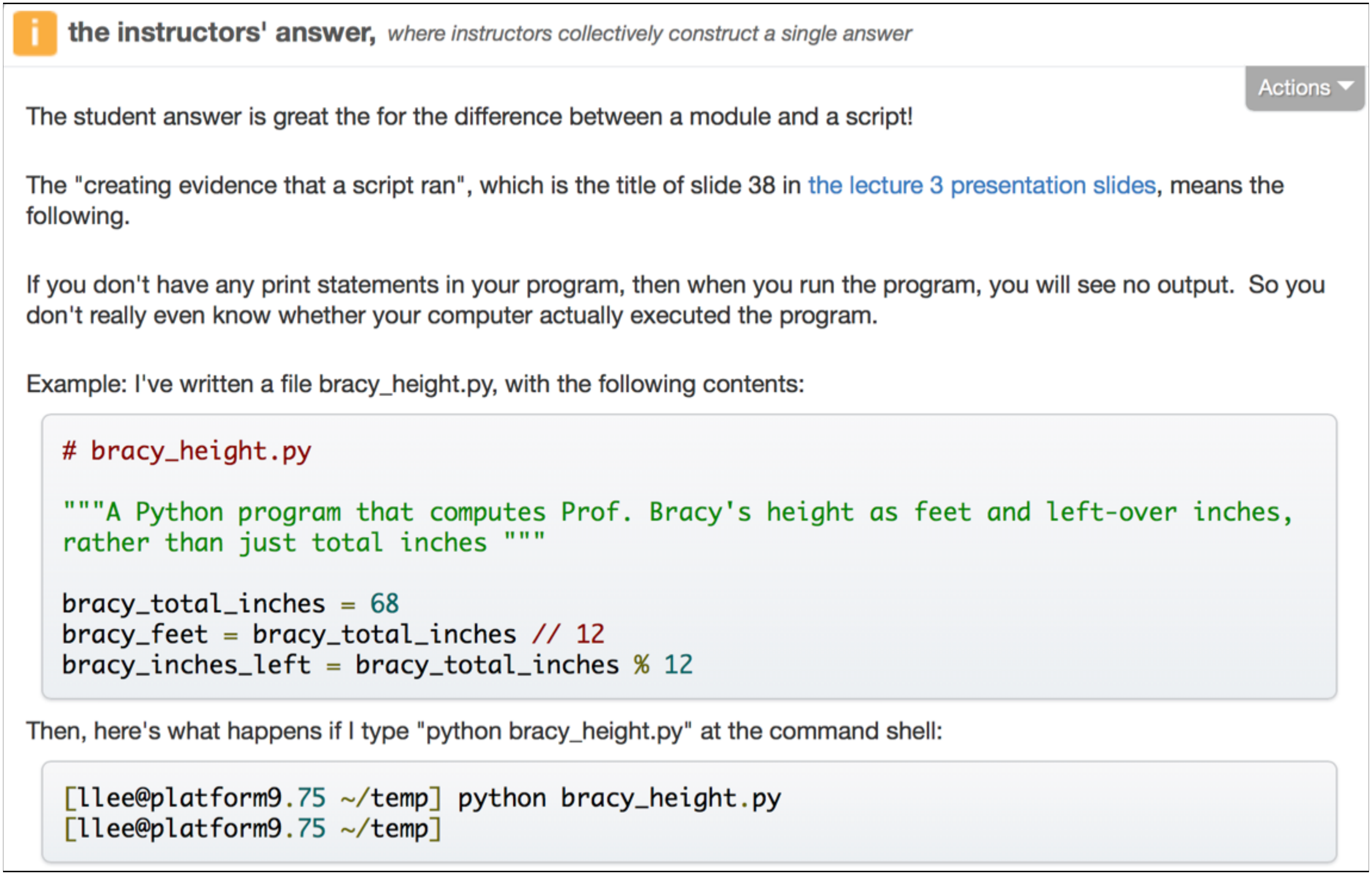
if so what is the interactive mode for a script?

#pin

other

~ An instructor (Prof. Lee) thinks this is a good question ~

https://piazza.com/class/jckqwmqflaz6i?cid=21



the instructors' answer, *where instructors collectively construct a single answer*

The student answer is great the for the difference between a module and a script!

The "creating evidence that a script ran", which is the title of slide 38 in the lecture 3 presentation slides, means the following.

If you don't have any print statements in your program, then when you run the program, you will see no output. So you don't really even know whether your computer actually executed the program.

Example: I've written a file bracy_height.py, with the following contents:

```
# bracy_height.py

"""A Python program that computes Prof. Bracy's height as feet and left-over inches,
rather than just total inches """

bracy_total_inches = 68
bracy_feet = bracy_total_inches // 12
bracy_inches_left = bracy_total_inches % 12
```

Then, here's what happens if I type "python bracy_height.py" at the command shell:

```
[llee@platform9.75 ~/temp] python bracy_height.py
[llee@platform9.75 ~/temp]
```

Anway, NOTHING seems to have happened. I have no evidence that Python did the int division or did any of my commands.
Similarly, here's what happens if I enter Python interactive mode, and then import the module:

```
[llee@platform9.75 ~/temp] python
Python 3.6.1 |Anaconda 4.4.0 (x86_64)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import bracy_height
>>>
```

Again, it looks like nothing happened.

So, if I want evidence that Python actually computed Prof. Bracy's height in feet and inches, I can force it to provide this evidence by asking it to print its results. So, here's a new program, bracy_height_verbose.py:

```
# bracy_height_verbose.py

"""A Python program that computes Prof. Bracy's height as feet and left-over inches,
rather than just total inches,
and prints the feet and left-over inches """

bracy_total_inches = 68
bracy_feet = bracy_total_inches // 12
bracy_inches_left = bracy_total_inches % 12

print("Prof. Bracy is " + str(bracy_feet) + " feet " + str(bracy_inches_left))
```

Now, if I run this as a script at the command shell, here's what happens:

```
[llee@platform9.75 ~/temp] python bracy_height_verbose.py
Prof. Bracy is 5 feet 8
[llee@platform9.75 ~/temp]
```

So you can see evidence that the script ran: it gave an output. Hurrah!

You can also treat the file as a module, and import it in Python interactive mode, as so:

```
[llee@platform9.75 ~/temp] python
Python 3.6.1 |Anaconda 4.4.0 (x86_64)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import bracy_height_verbose
Prof. Bracy is 5 feet 8
>>>
```

And again, you can see that the module does something, in this case, because it gave an output. Hurrah again!

From last time: Function Calls

- Function expressions have the form $\underbrace{\textbf{fun}}_{\text{function name}}(\underbrace{x}_{\text{argument}},y,\ldots)$

- **Examples** (math functions that work in Python):
  - `round(2.34)`
  - `max(a+3,24)`

From last time: Modules

- Modules provide extra functions, variables
  - Access them with the `import` command
- **Example**: module `math`

```
>>> import math
>>> math.cos(2.0)
-0.4161468365471424
>>> math.pi
3.141592653589793
```

From last time: Modules

Module Text	Interactive Python

```
# my_module.py

"""This is a simple module.
It shows how modules work"""

x = 1+2
x = 3*x
```

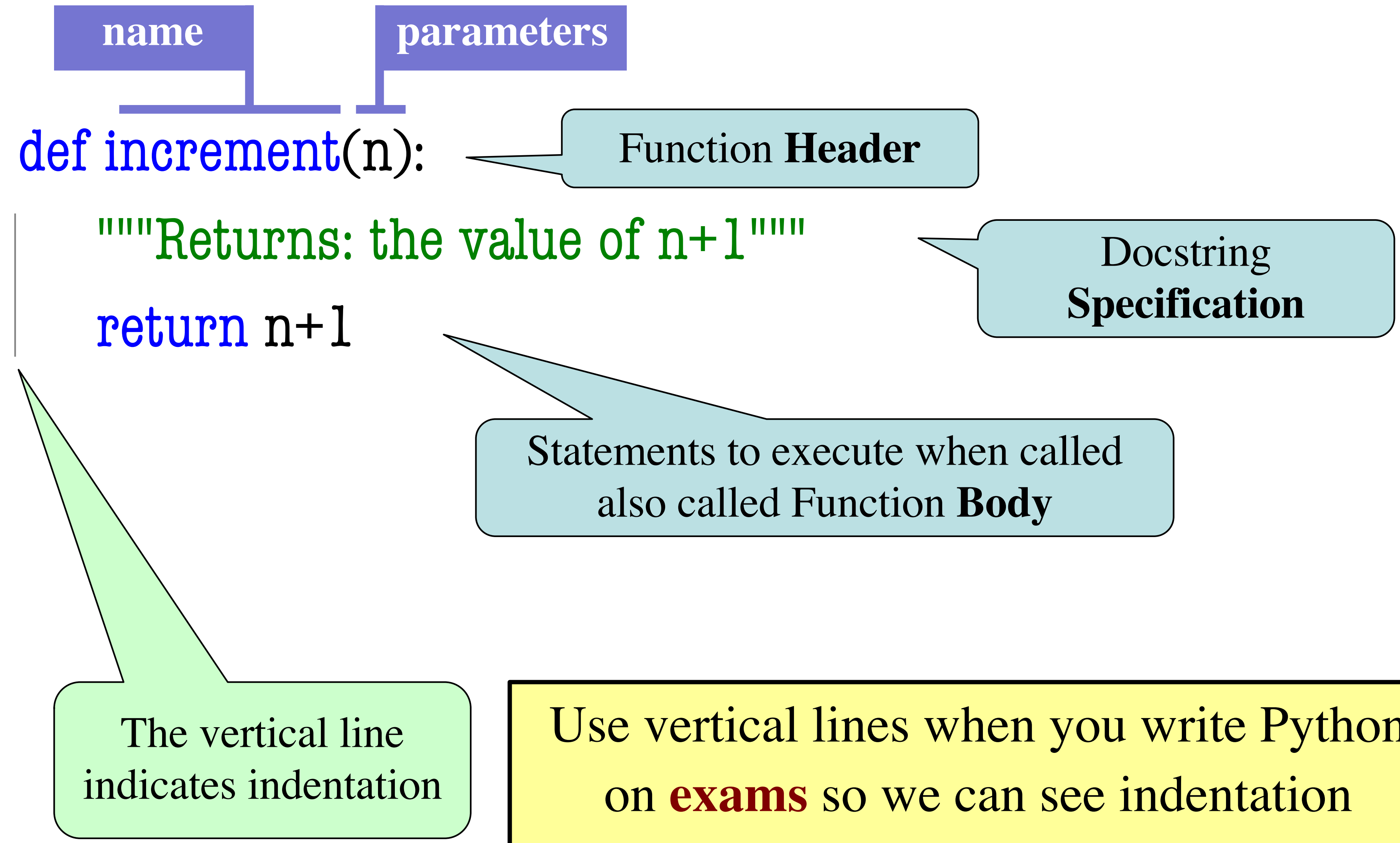
```
>>> import my_module
>>> my_module.x
9
```

- We discussed how to make module *variables*
- Have not covered how to make *functions*

simple_math.py

```
>>> import simple_math
>>> simple_math.increment(1)
2
>>> simple_math. increment(2)
3
```

Anatomy of a Function Definition



```python
def increment(n):
    """Returns: the value of n+1"""
    return n+1
```

name
parameters
Function **Header**
Docstring **Specification**
Statements to execute when called also called Function **Body**
The vertical line indicates indentation

Use vertical lines when you write Python on **exams** so we can see indentation

The **`return`** Statement

- Passes a value from the function to the caller
- **Format**: **`return`** *<expression>*
- Any statements after **`return`** are ignored
- Optional (if absent, special value **`None`** will be sent back)

Function Calls vs. Definitions

Function Call

- Command to **do** the function

```
>>> simple_math.increment(23)
24
>>>
```

argument to assign to n

Function Definition

- Defines what function **does**

```
def increment(n):
    return n+1
```

declaration of **parameter** n

- **Parameter**: variable that is listed within the parentheses of a function header.
- **Argument**: a value to assign to the function parameter when it is called

Using simple_math.py

Module Text

```
# simple_math.py

"""module with two simple
   math functions"""

def increment(n):
   """Returns: n+1"""
   return n+1
```

Python skips

Python skips

Python learns the function definition

Python skips everything inside the function

Repeat for all functions in module

Interactive Python

```
>>> import simple_math
```

Using simple_math.py

Module Text

```
# simple_math.py

"""module with two simple
   math functions"""

def increment(n):
   """Returns: n+1"""
   return n+1
```

Now Python executes the function body

Interactive Python

```
>>> import simple_math
>>> simple_math.increment(23)
```

Python knows what this is!

Using **simple_math.py**

Module Text

```
# simple_math.py

"""module with two simple
   math functions"""


def increment(n):
    """Returns: n+1"""
    return n+1
```

Interactive Python

```
>>> import simple_math
>>> simple_math.increment(23)
```

Using **simple_math.py**

Module Text

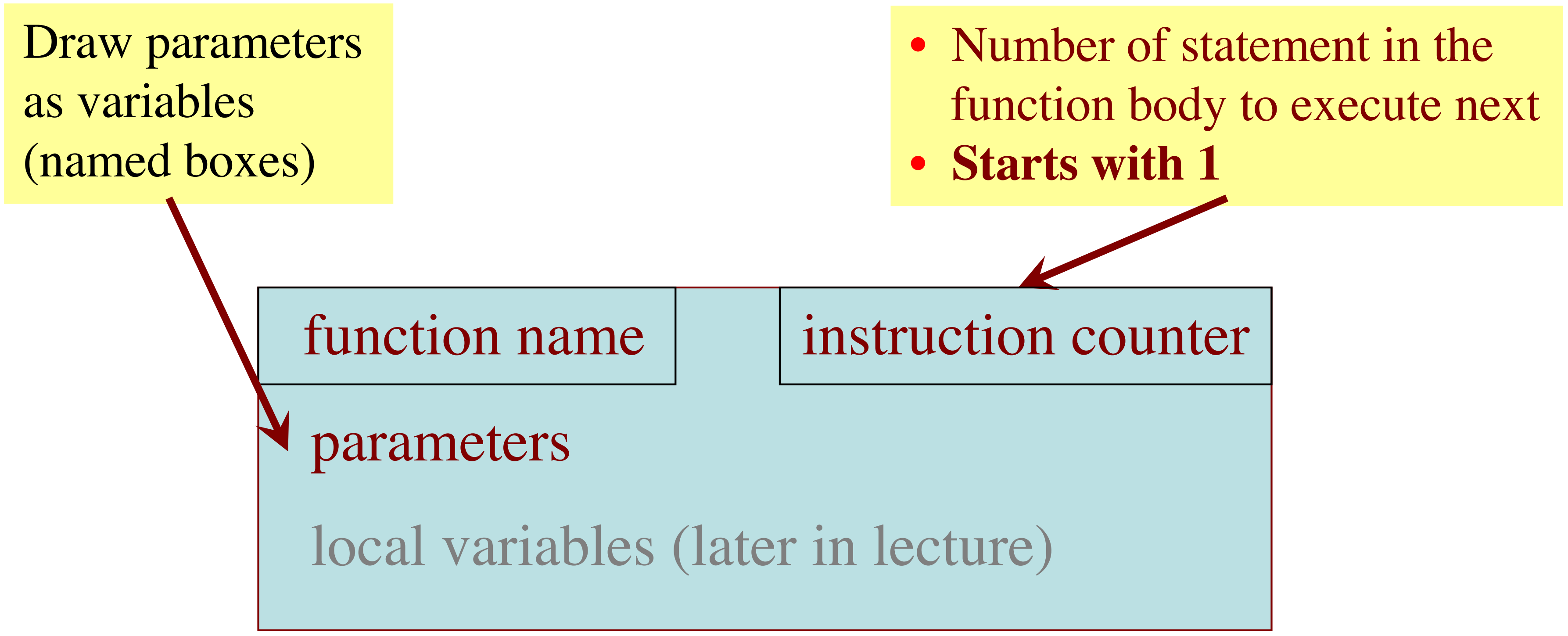
```python
# simple_math.py

"""module with two simple
   math functions"""


def increment(n):
   """Returns: n+1"""
   return n+1
```

Interactive Python

```
>>> import simple_math
>>> simple_math.increment(23)
24
```

Understanding How Functions Work

- We will draw pictures to show what is in memory
- **Function Frame**: Representation of function call


Draw parameters as variables (named boxes)
Number of statement in the function body to execute next
Starts with 1
function name
instruction counter
parameters
local variables (later in lecture)

Note: slightly different than in the book (3.9) Please do it **this** way.

Example: get_feet

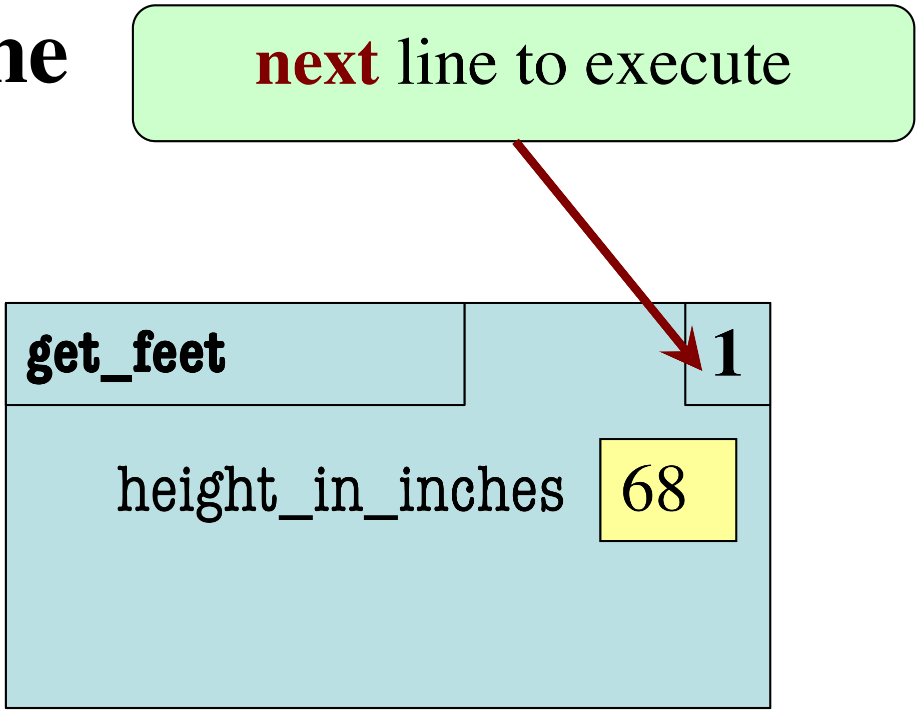
```
>>> import height
>>> height.get_feet(68)
```

```
def get_feet(height_in_inches):
    return height_in_inches // INCHES_PER_FOOT
```

Example: get_feet(68)

PHASE 1: Set up call frame

1. Draw a frame for the call
2. Assign the argument value to the parameter (in frame)
3. Indicate next line to execute

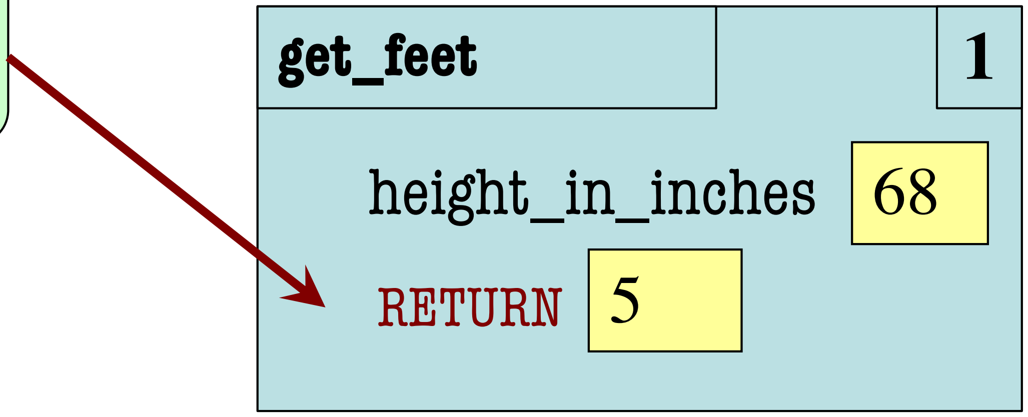


```
  def get_feet(height_in_inches):
1     return height_in_inches // INCHES_PER_FOOT
```

Example: get_feet(68)

PHASE 2:
Execute function body

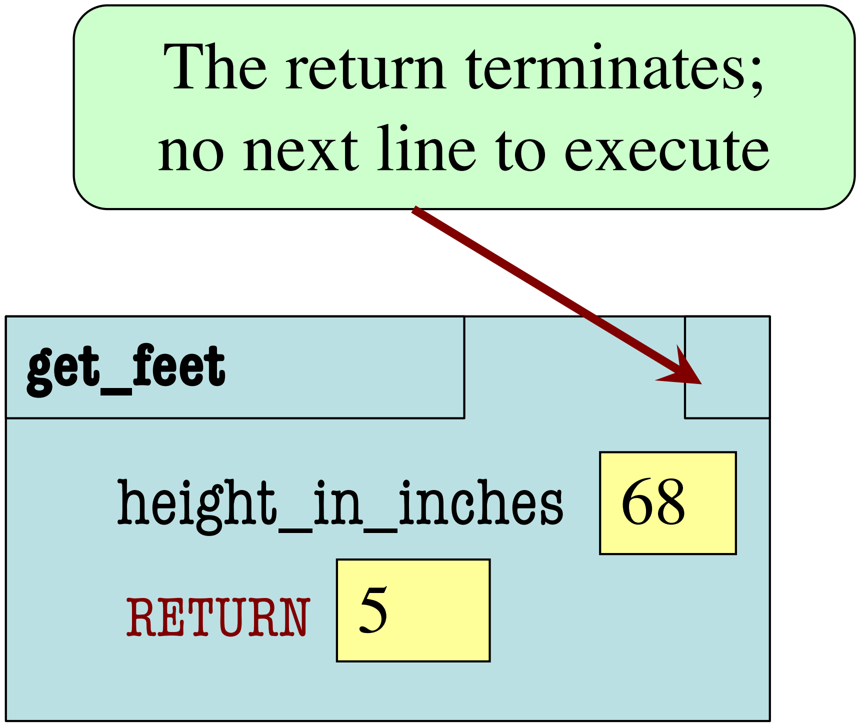
Return statement creates a special variable for result



```
    def get_feet(height_in_inches):
1       return height_in_inches // INCHES_PER_FOOT
```

Example: get_feet(68)

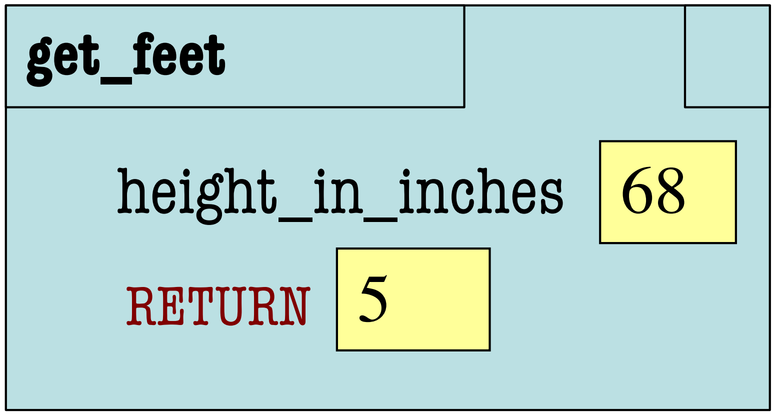
PHASE 2:
Execute function body



```
def get_feet(height_in_inches):
    return height_in_inches // INCHES_PER_FOOT
```

Example: get_feet(68)

PHASE 3: Erase call frame



```
def get_feet(height_in_inches):
    return height_in_inches // INCHES_PER_FOOT
```

Example: get_feet(68)

PHASE 3: Erase call frame

But don't actually erase on an exam

ERASE WHOLE FRAME

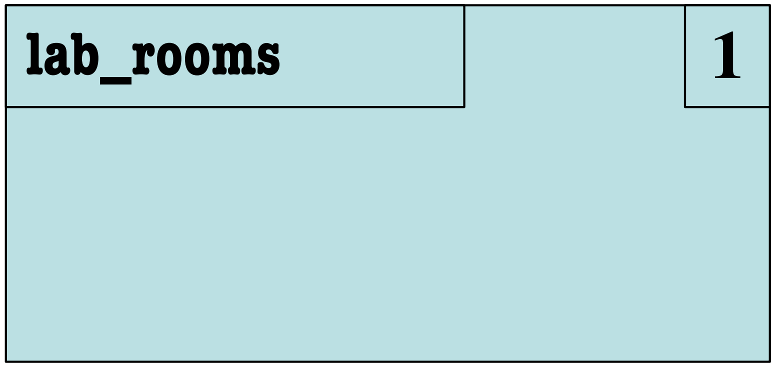
```
def get_feet(height_in_inches):
    return height_in_inches // INCHES_PER_FOOT
```

Local Variables (1)

- Call frames can make "local" variables

```
>>> import room_numbers
>>> room_numbers.lab_rooms()
```

```
  def lab_rooms():
1     red_room = 235
2     orange_room = 236
```

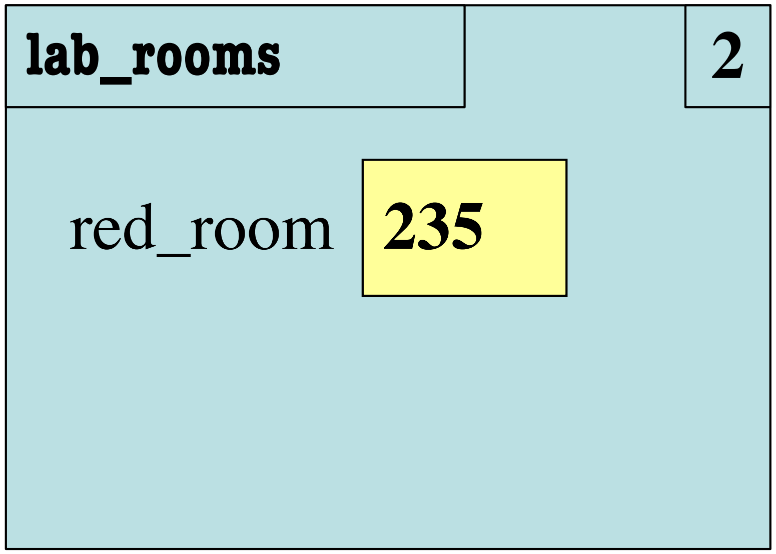


Local Variables (2)

- Call frames can make "local" variables

```
>>> import room_numbers
>>> room_numbers.lab_rooms()
```

```
  def lab_rooms():
1     red_room = 235
2     orange_room = 236
```

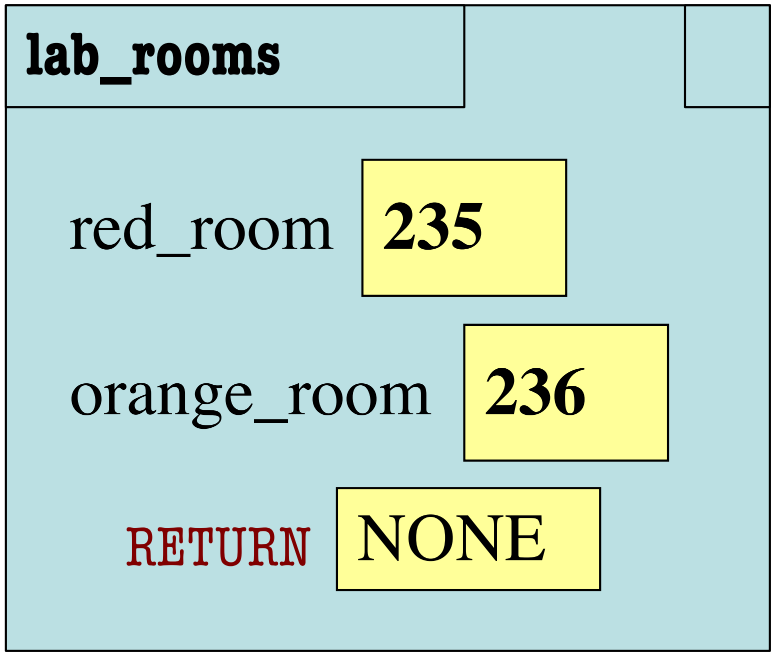


Local Variables (3)

- Call frames can make “local” variables

```
>>> import room_numbers
>>> room_numbers.lab_rooms()
```

```
  def lab_rooms():
1     red_room = 235
2     orange_room = 236
```



Local Variables (4)

- Call frames can make “local” variables

```
>>> import room_numbers
>>> room_numbers.lab_rooms()
```

ERASE WHOLE FRAME

```
def lab_rooms():
1    red_room = 235
2    orange_room = 236
```

Variables are gone! This function is useless.

Exercise Time

Function Definition

```
def foo(a,b):
    x = a
    y = b
    return x*y+y
```

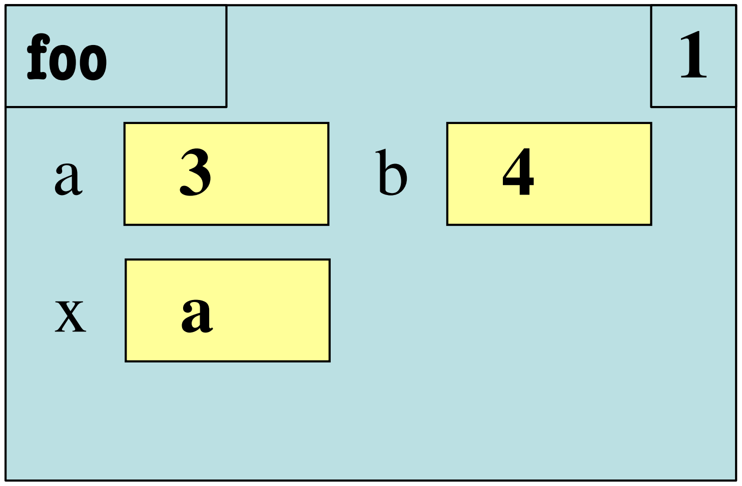
Function Call

```
>>> foo(3,4)
```

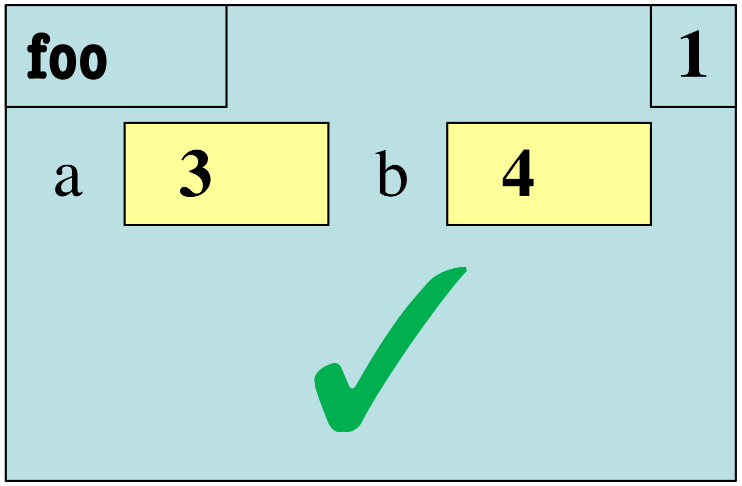
What does the frame look like at the **start**?

Which One is Closest to Your Answer?

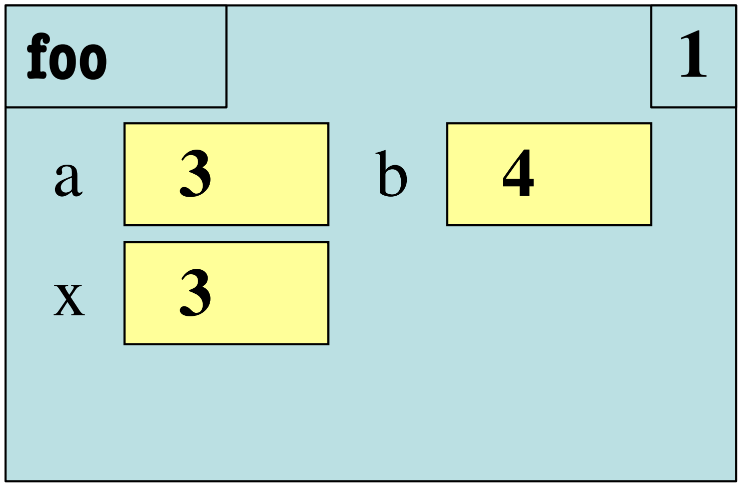
A:



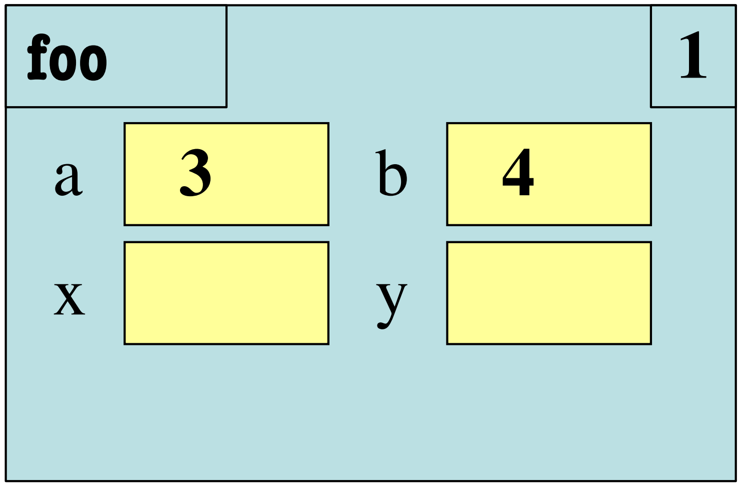
B:



C:



D:



Exercise Time

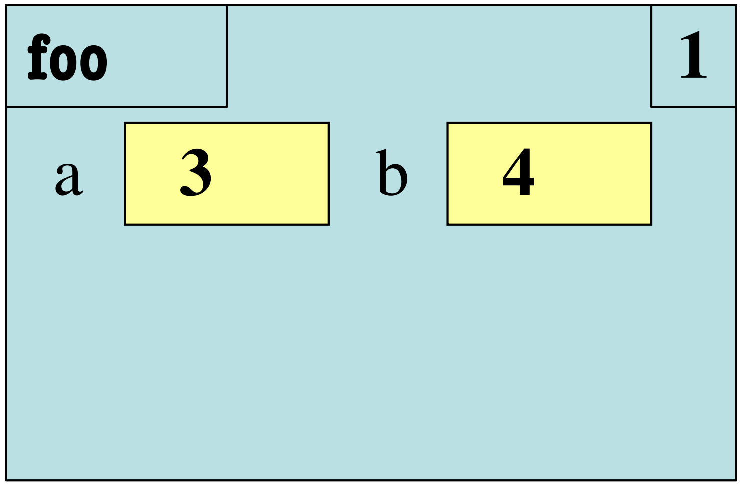
Function Definition

```
def foo(a,b):
1     x = a
2     y = b
3     return x*y+y
```

Function Call

```
>>> foo(3,4)
```

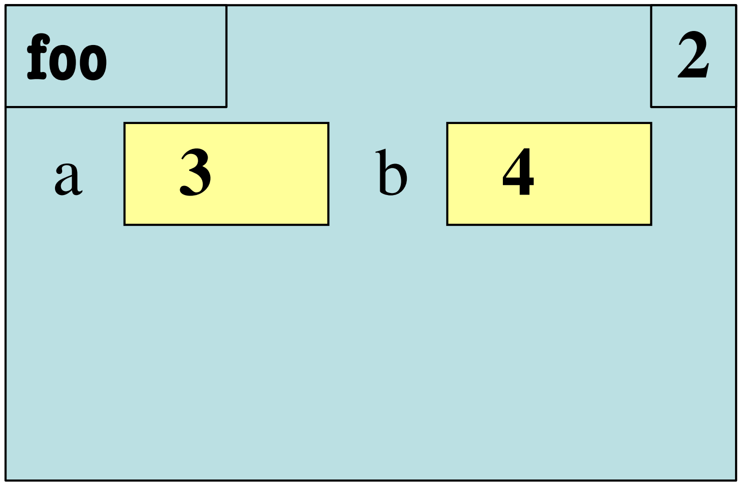
B:



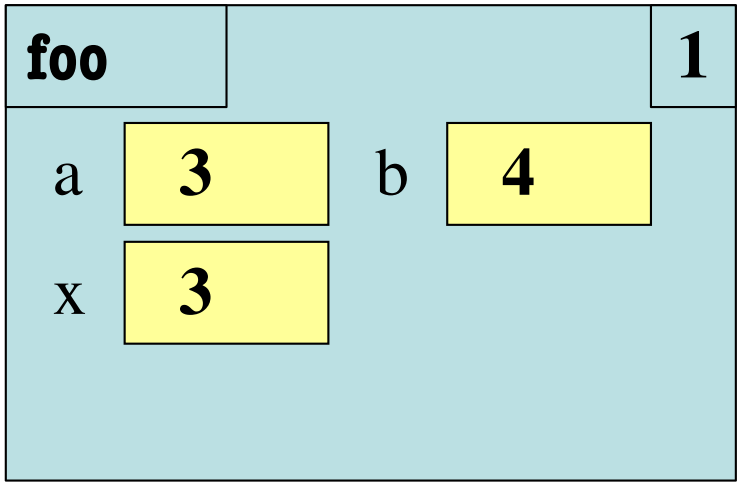
What is the **next step**?

Which One is Closest to Your Answer?

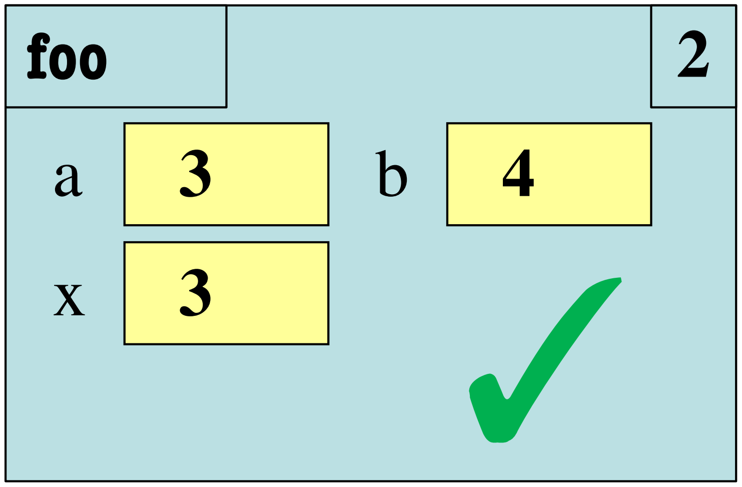
A:



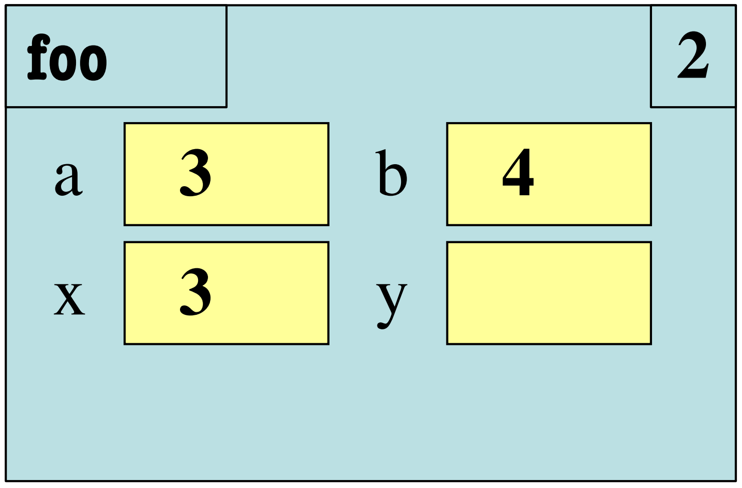
B:



C:



D:



Exercise Time

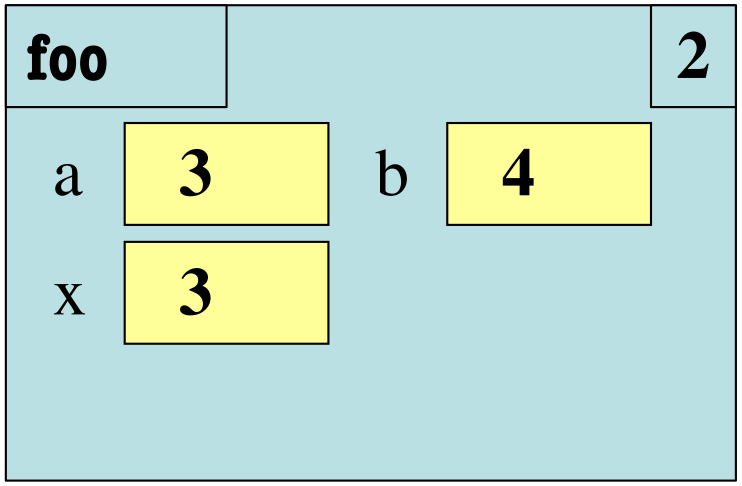
Function Definition

```
def foo(a,b):
1     x = a
2     y = b
3     return x*y+y
```

Function Call

```
>>> foo(3,4)
```

C:



What is the **next step**?

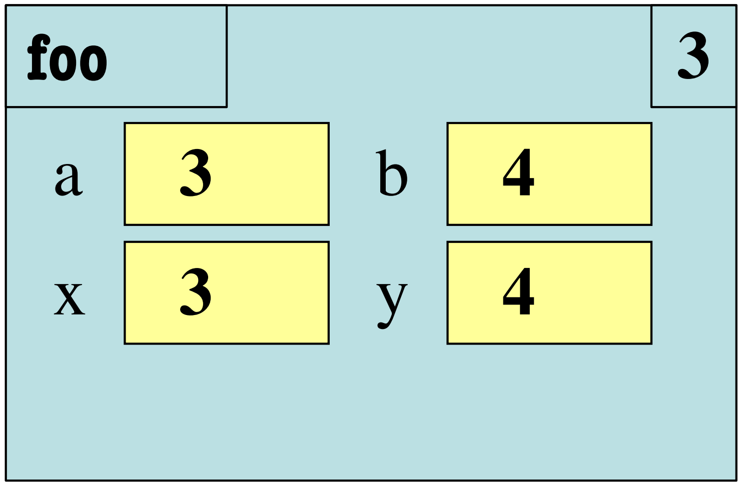
Exercise Time

Function Definition

```
def foo(a,b):
1     x = a
2     y = b
3     return x*y+y
```

Function Call

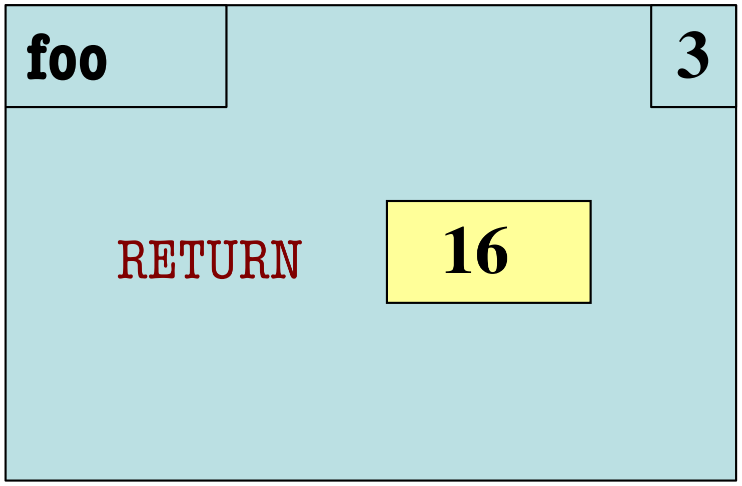
```
>>> foo(3,4)
```



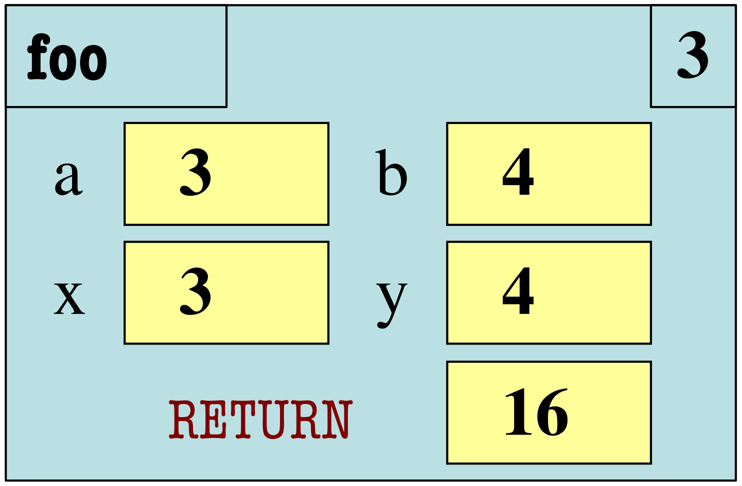
What is the **next step**?

Which One is Closest to Your Answer?

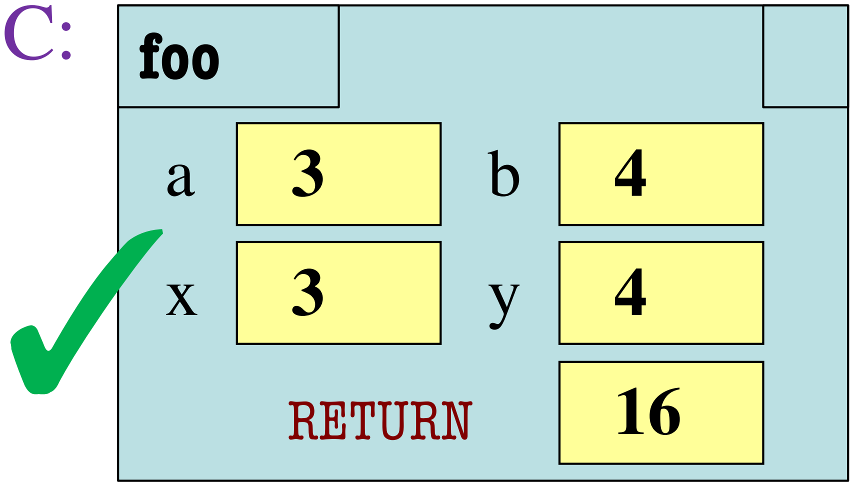
A:



B:



C:



D:

ERASE THE FRAME

Exercise Time

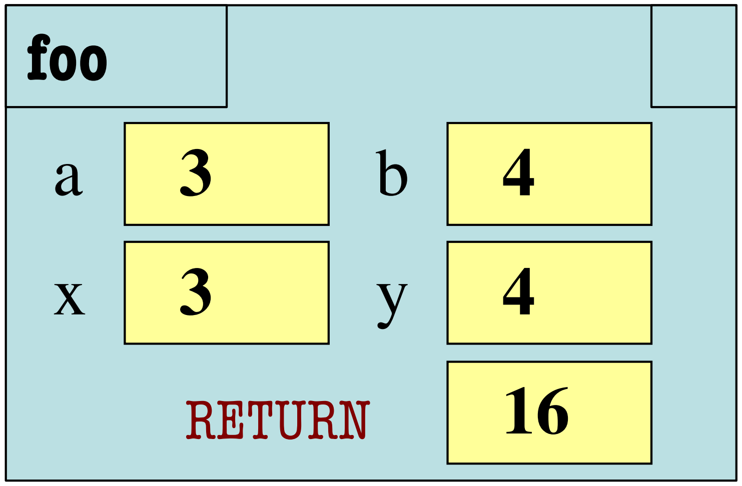
Function Definition

```
def foo(a,b):
1    x = a
2    y = b
3    return x*y+y
```

Function Call

```
>>> foo(3,4)
```

C:



What is the **next step**?

Exercise Time

Function Definition

```
def foo(a,b):
    x = a
    y = b
    return x*y+y
```

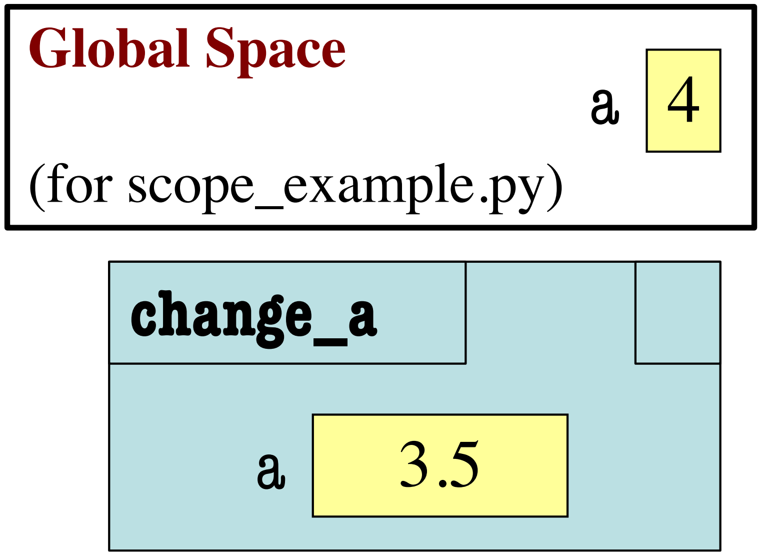
Function Call

```
>>> foo(3,4)
>>> 16
```

ERASE THE FRAME

Function Access to Global Space

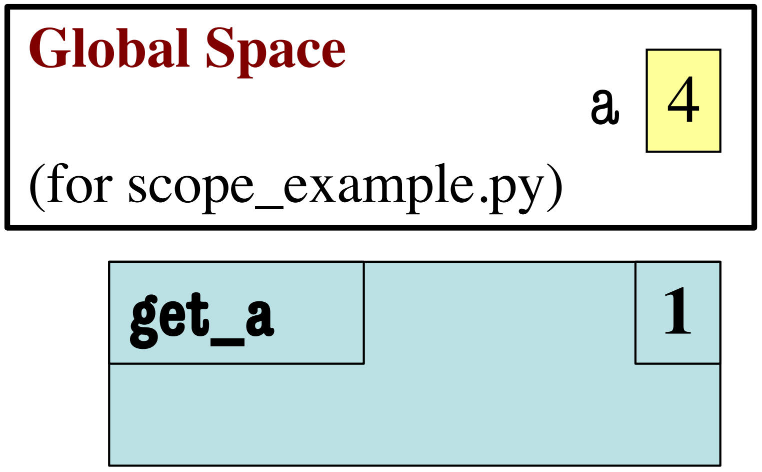
- All function definitions are in some module
- Call can access global space for **that module**
  - math.cos: global for math
  - height.get_feet uses global for height
- But **cannot** change values
  - "Assignment to a global" makes a new local variable!



```
# scope_example.py
"""Show how globals work"""
a = 4 # global space
def change_a():
    a = 3.5 # local variable
    return a
```

Function Access to Global Space

- All function definitions are in some module
- Call can access global space for **that module**
  - math.cos: global for math
  - height.get_feet uses global for height
- But **cannot** change values
  - Assignment to a global makes a new local variable!



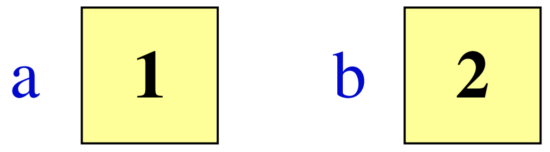
```
# scope_example.py
"""Show how globals work"""
a = 4 # global space

def get_a():
    return a # returns global
```

Call Frames and Global Variables

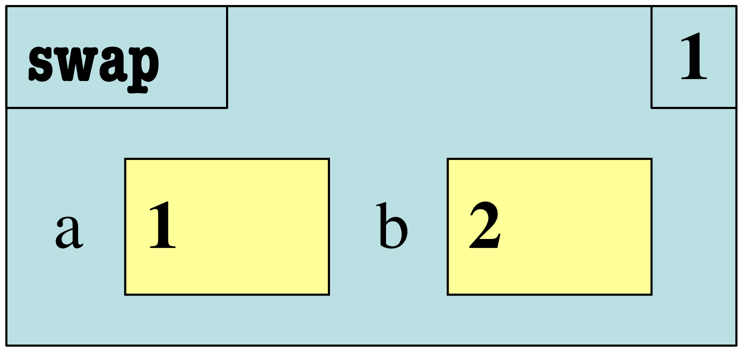
```
  def swap(a,b):
      """Swap global a & b"""
1     tmp = a
2     a = b
3     b = tmp
```

```
>>> a = 1
>>> b = 2
>>> swap(a,b)
```

Global Variables



Call Frame

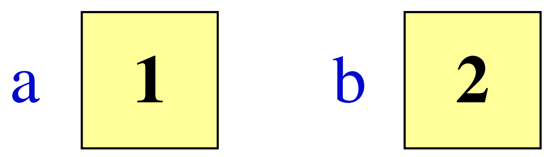


Call Frames and Global Variables

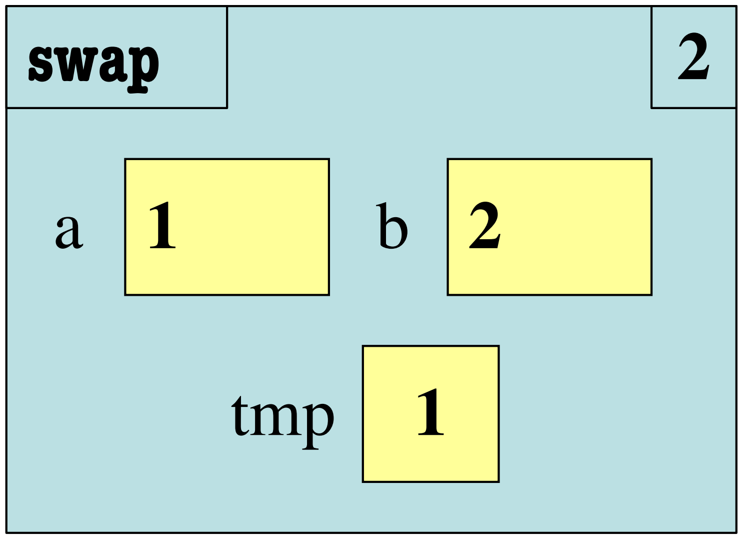
```
   def swap(a,b):
       """Swap global a & b"""
1      tmp = a
2      a = b
3      b = tmp
```

```
>>> a = 1
>>> b = 2
>>> swap(a,b)
```

Global Variables



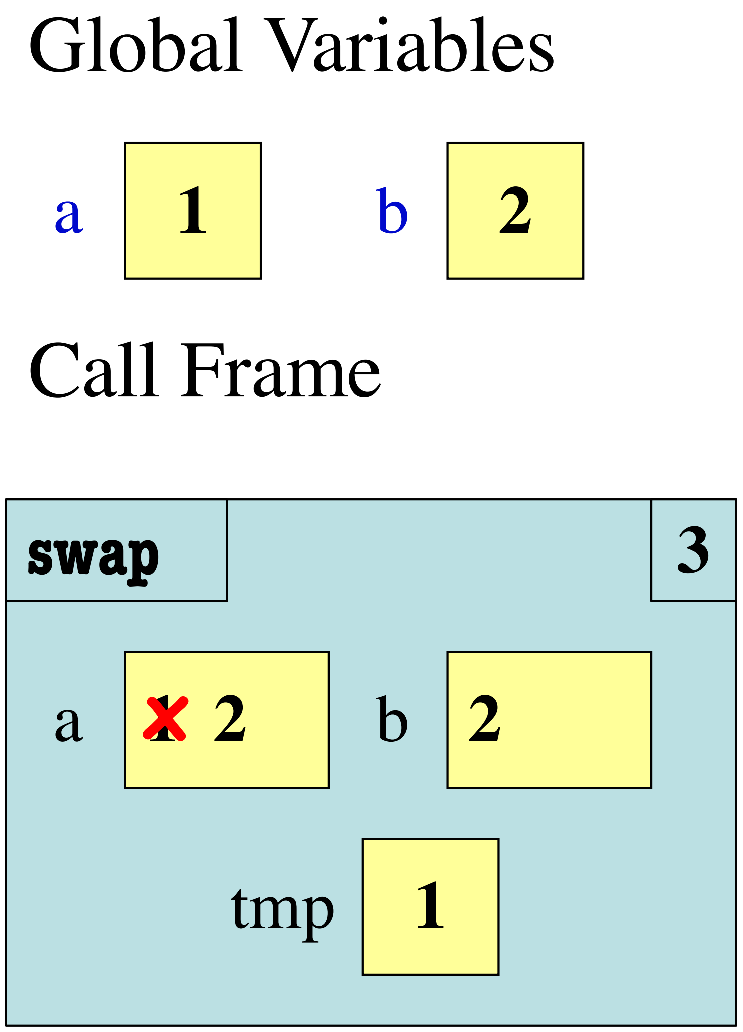
Call Frame



Call Frames and Global Variables

```
  def swap(a,b):
      """Swap global a & b"""
1     tmp = a
2     a = b
3     b = tmp
```

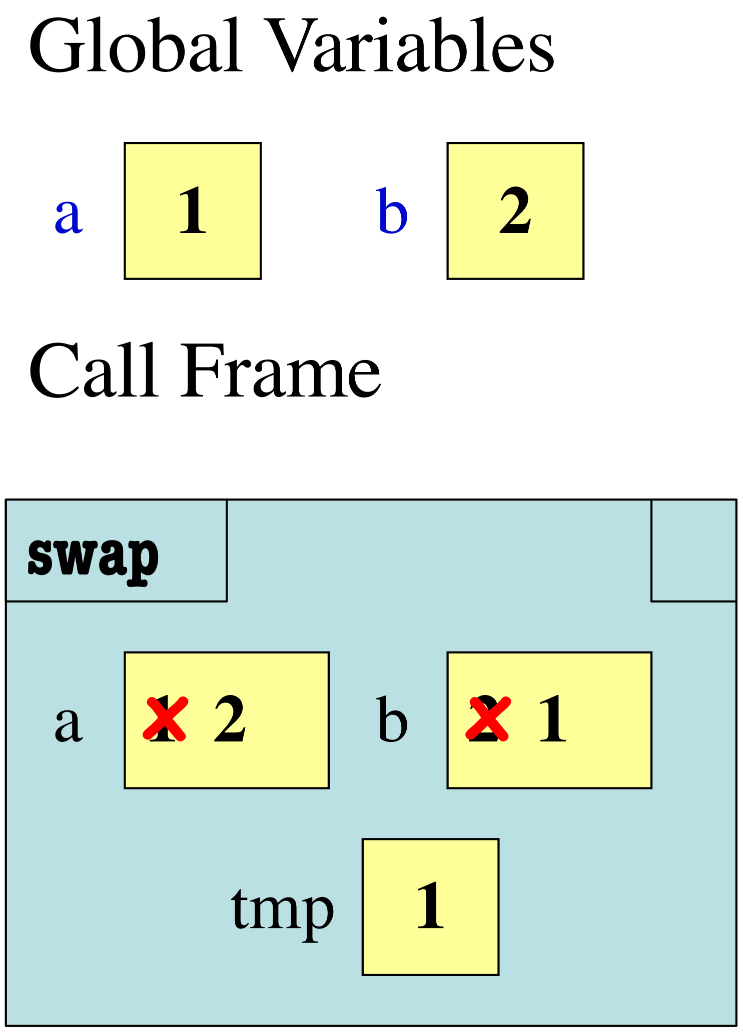
```
>>> a = 1
>>> b = 2
>>> swap(a,b)
```



Call Frames and Global Variables

```
def swap(a,b):
    """Swap global a & b"""
1   tmp = a
2   a = b
3   b = tmp
```

```
>>> a = 1
>>> b = 2
>>> swap(a,b)
```

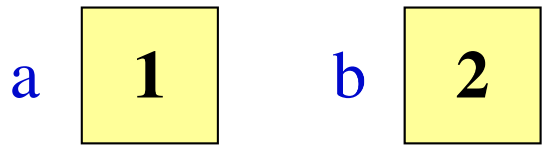


Call Frames and Global Variables

```
def swap(a,b):
    """Swap global a & b"""
    tmp = a
    a = b
    b = tmp
```

```
>>> a = 1
>>> b = 2
>>> swap(a,b)
```

Global Variables



Call Frame

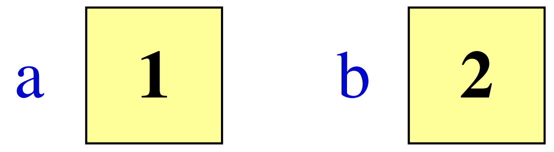
ERASE THE FRAME

Call Frames and Global Variables

```
def swap(a,b):
    """Swap global a & b"""
    tmp = a
    a = b
    b = tmp
```

```
>>> a = 1
>>> b = 2
>>> swap(a,b)
```

Global Variables

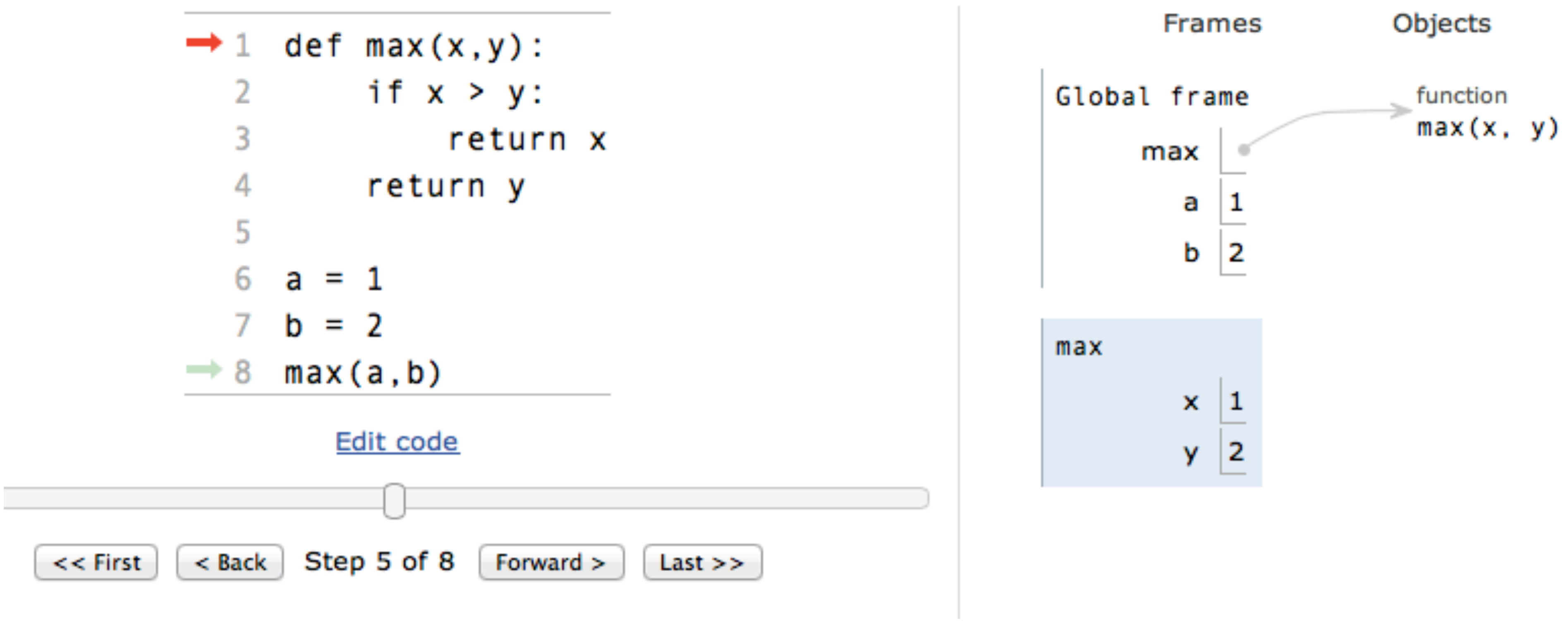


Call Frame

RAME

THIS FUNCTION DOES NOT SWAP the *global* a and *global* b

Visualizing Frames: The Python Tutor



```
def max(x,y):
    if x > y:
        return x
    return y

a = 1
b = 2
max(a,b)
```

More Exercises

Module Text

```
# my_module.py

def foo(x):
    return x+1

x = 1+2
x = 3*x
```

Interactive Python

```
>>> import my_module
>>> my_module.x
...
```

What does Python give me?

A: 9 **CORRECT**
B: 10
C: 1
D: Nothing
E: Error