http://www.cs.cornell.edu/courses/cs1110/2020sp

Lecture 4: Defining Functions

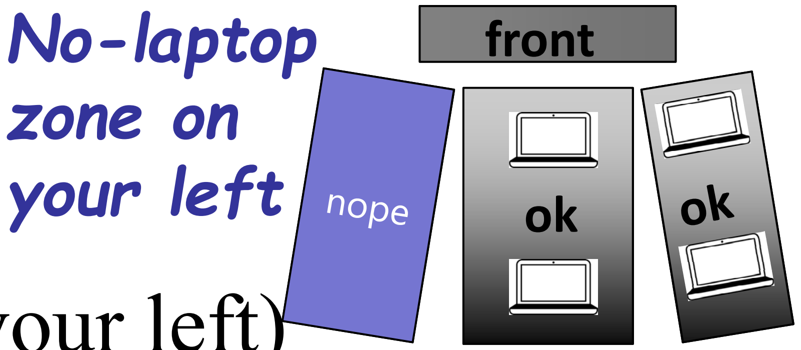
(Ch. 3.4-3.11)

CS 1110

Introduction to Computing Using Python

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

Announcements



- No laptop use stage right (your left)
- We will use clickers, but not for credit. Therefore no need to register your clicker.
- "Partner Finding Social" Tues Feb 4th 5-6pm Gates Hall 3rd floor Lounge (1xxx-2xxx courses)
- Before next lecture, read Sections 8.1, 8.2, 8.4, 8.5, 1st ¶ of 8.9

Review ideas from Lecture 2 & Lab 2

Module vs. Script

`print` statement

Clicker Question

my_module.py

```
# my_module.py

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

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

Command Line

```
C:\> python my_module.py
C:\> my_module.x
```

After you hit “Return” here what will be printed next?

(A) >>>

(B) 9
>>>

(C) an error message

(D) The text of my_module.py

(E) Sorry, no clue.

Clicker Answer

my_module.py

```python
# my_module.py

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

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

Command Line

```
C:\> python my_module.py
C:\> my_module.x
```

After you hit "Return" here what will be printed next?

(A) >>>

(B) 9
     >>>

[✓ (C) an error message]

(D) The text of my_module.py

(E) Sorry, no clue.

Running my_module.py as a script

my_module.py

```
# my_module.py

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

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

Command Line

```
C:\> python my_module.py
C:\>
```

when the script ends, all memory used by my_module.py is deleted

thus, all variables get deleted (including x)

so there is no evidence that the script ran

my_module.py vs. script.py

my_module.py

```
# my_module.py

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

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

script.py

```
# script.py

""" This is a simple script.
It shows why we use print"""

x = 1+2
x = 3*x
print(x)
```

Only difference

Syntax:
`print (<expression>)`

Running script.py as a script

Command Line

```
C:\> python script.py
9


C:\>
```

script.py

```
# script.py

""" This is a simple script.
It shows why we use print"""

x = 1+2
x = 3*x
print(x)
```

Modules vs. Scripts

Module	Script
• Provides functions, variables	• Behaves like an application
• import it into Python shell	• Run it from command line

Files could look the same.
Difference is how you use them.

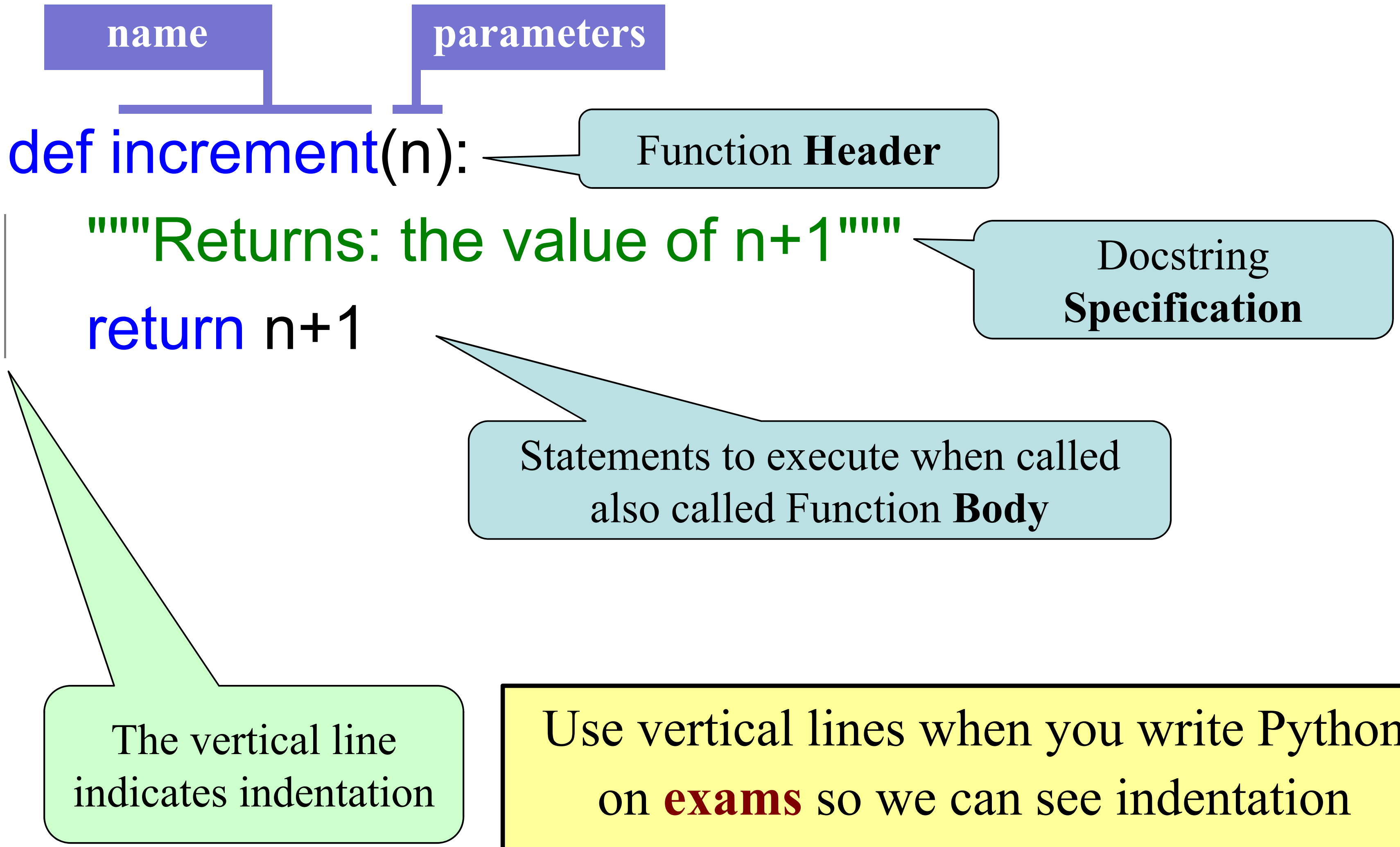
Defining our own functions

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)

Let’s define our own functions!

Anatomy of a Function Definition


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

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

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 Definitions vs. Calls

```python
# simple_math.py


def increment(n):
    return n+1



increment(2)
```

simple_math.py

Function definition

- Defines what function **does**
- Declaration of parameter **n**
- **Parameter:** the variable that is listed within the parentheses of a function header.

Function call

- Command to do the function
- Argument to assign to **n**
- Argument: a value to assign to the function parameter when it is called

Executing the script simple_math.py

```
# simple_math.py

"""script that defines
and calls one simple
math function"""


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

increment(2)
```

Python skips

Python skips

Python learns about the function

Python skips everything inside the function

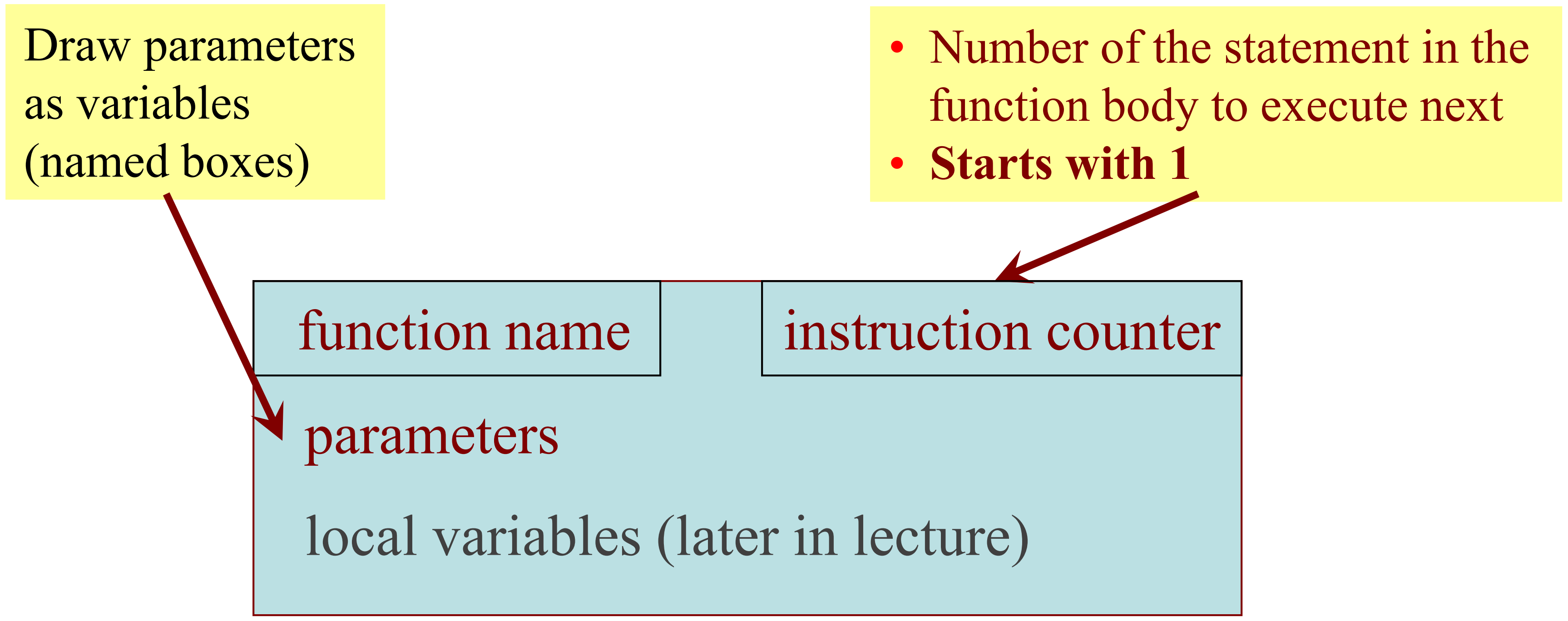
Python executes this statement

Now Python executes the function body

```
C:/> python simple_math.py
```

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 the 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 in height.py module

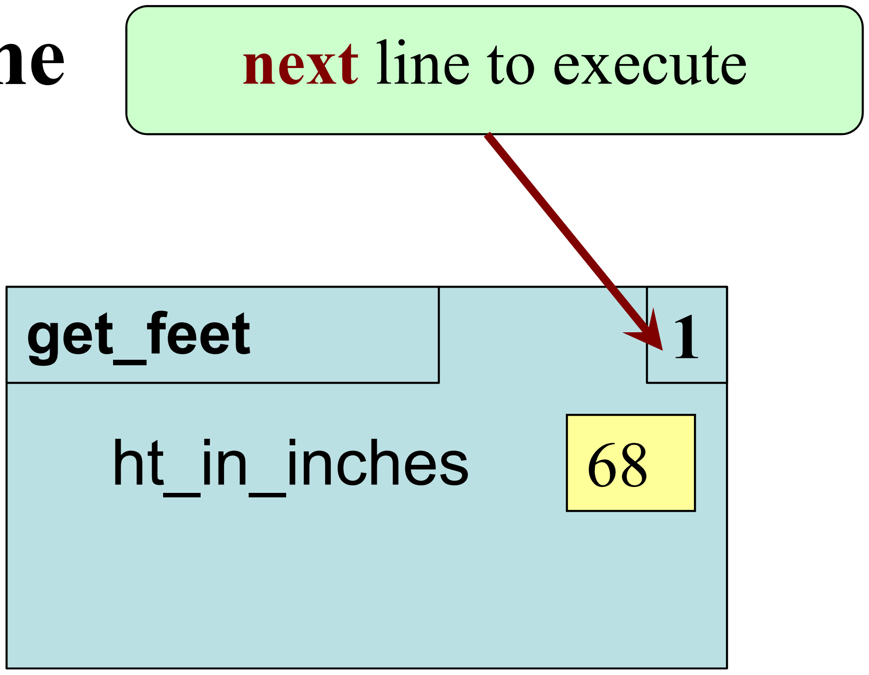
```
>>> import height
>>> height.get_feet(68)
```

```
def get_feet(ht_in_inches):
    return ht_in_inches // 12
```

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



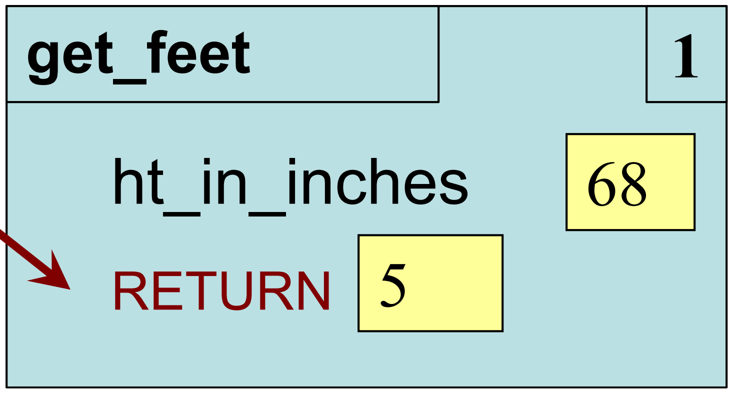
```
  def get_feet(ht_in_inches):
1     return ht_in_inches // 12
```

Example: get_feet(68)

PHASE 2:

Execute function body

Return statement creates a special variable for result

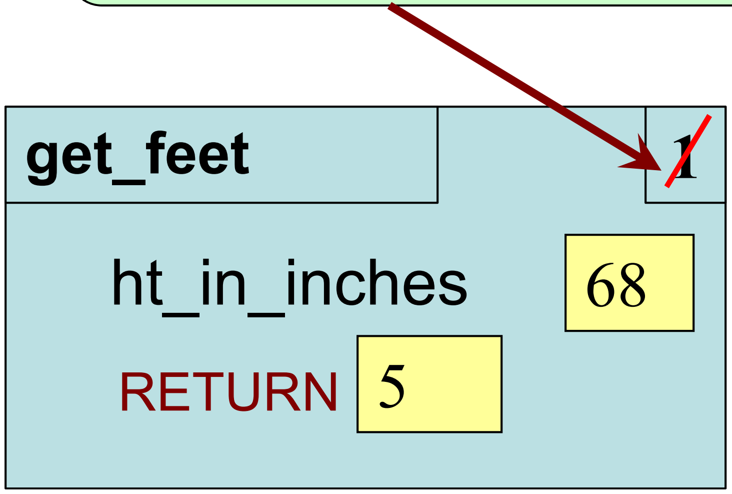


```
def get_feet(ht_in_inches):
1   return ht_in_inches // 12
```

Example: get_feet(68)

PHASE 2:
Execute function body

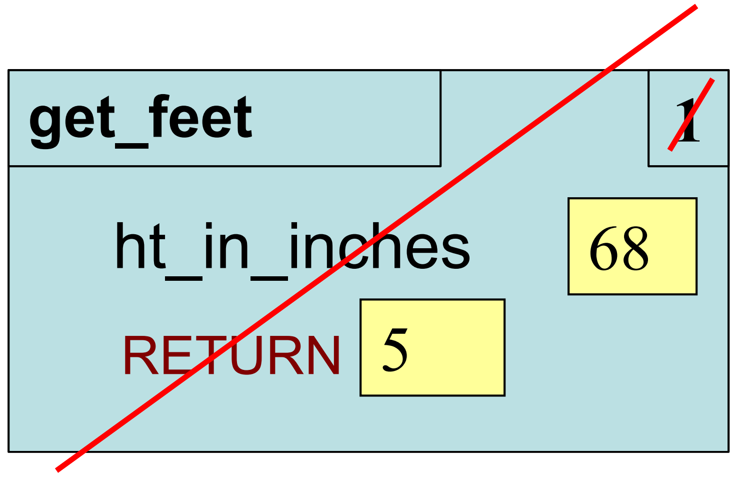
The return terminates; no next line to execute



```
def get_feet(ht_in_inches):
1    return ht_in_inches // 12
```

Example: get_feet(68)

PHASE 3: Delete (cross out) call frame



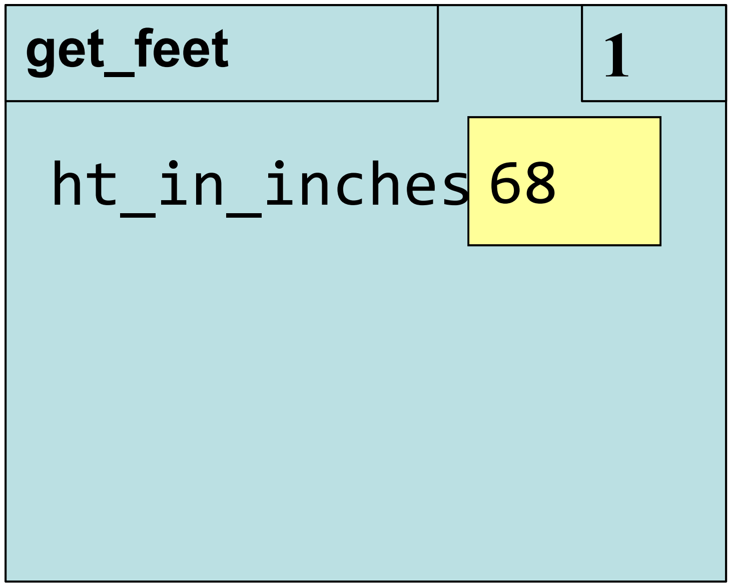
```
def get_feet(ht_in_inches):
    return ht_in_inches // 12
```

Local Variables (1)

- Call frames can make “local” variables
  - A variable created in the function

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

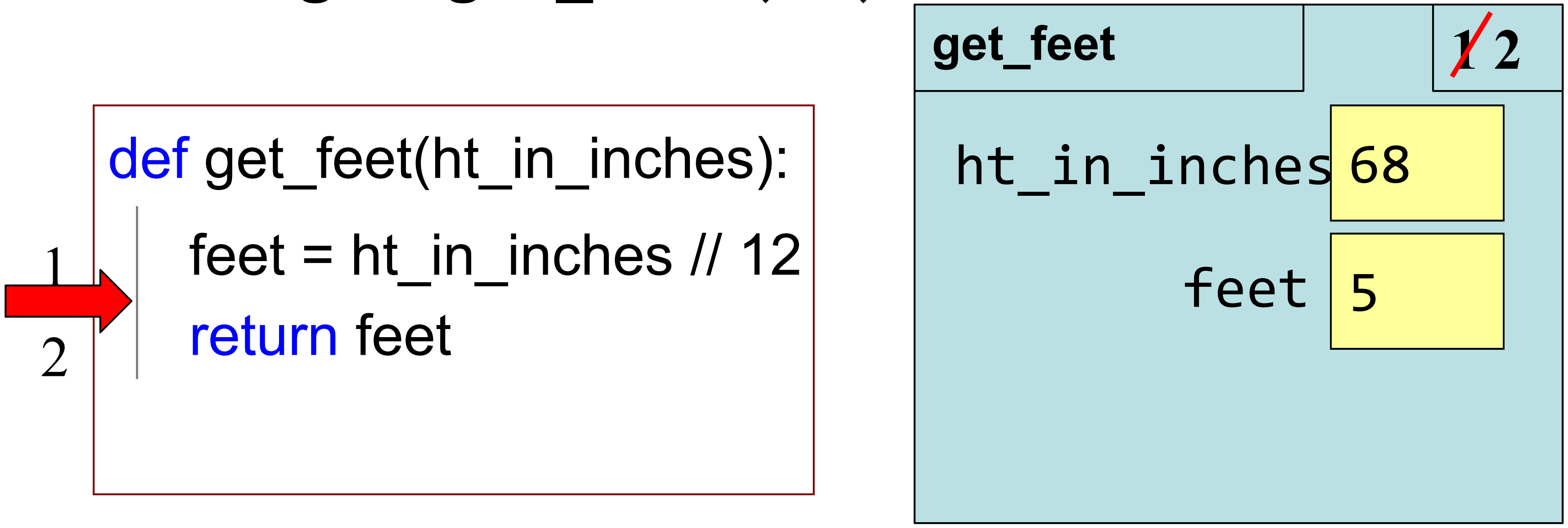
```
  def get_feet(ht_in_inches):
1     feet = ht_in_inches // 12
2     return feet
```



Local Variables (2)

- Call frames can make “local” variables
  - A variable created in the function

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



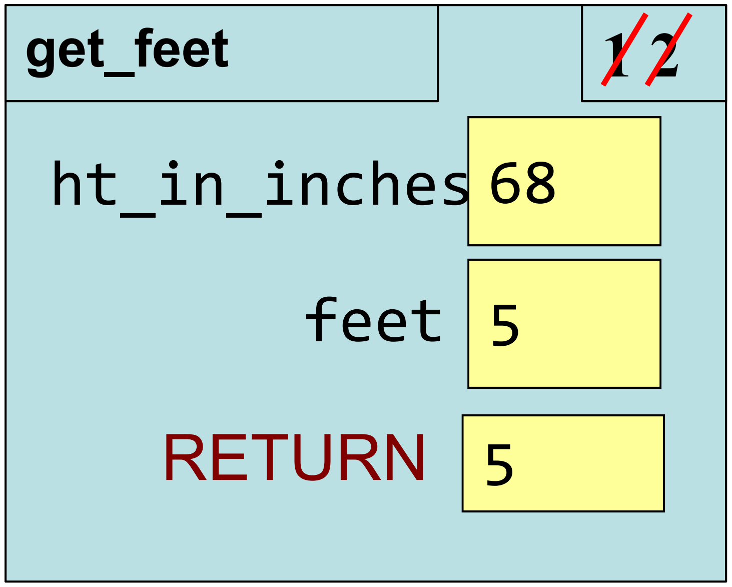
```
  def get_feet(ht_in_inches):
1     feet = ht_in_inches // 12
2     return feet
```

Local Variables (3)

- Call frames can make "local" variables
  - A variable created in the function

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

```
  def get_feet(ht_in_inches):
1     feet = ht_in_inches // 12
2     return feet
```

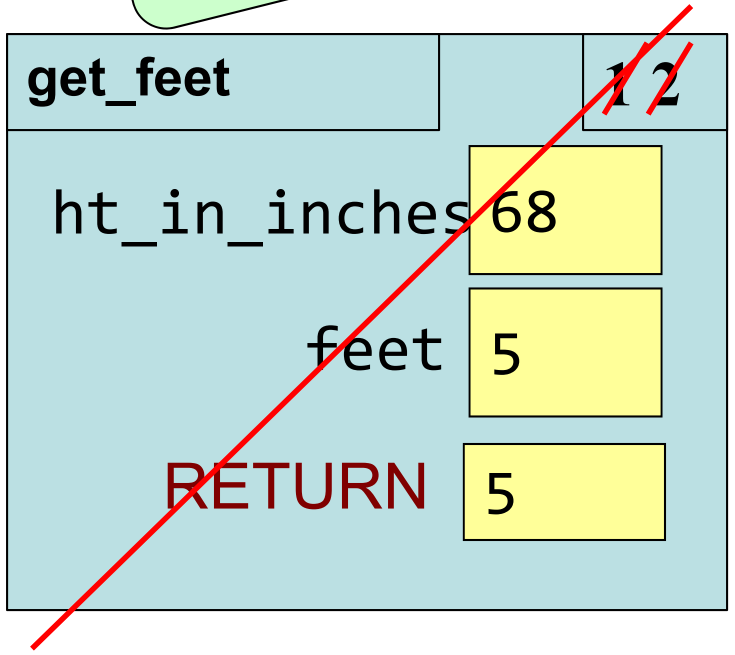


Local Variables (4)

- Call frames can make “local” variables
  - A variable created in the function

```
>>> import height
>>> height.get_feet(68)
>>> 5
```

```
  def get_feet(ht_in_inches):
1     feet = ht_in_inches // 12
2     return feet
```



Variables are gone! This function is over.

Exercise Time

Function Definition

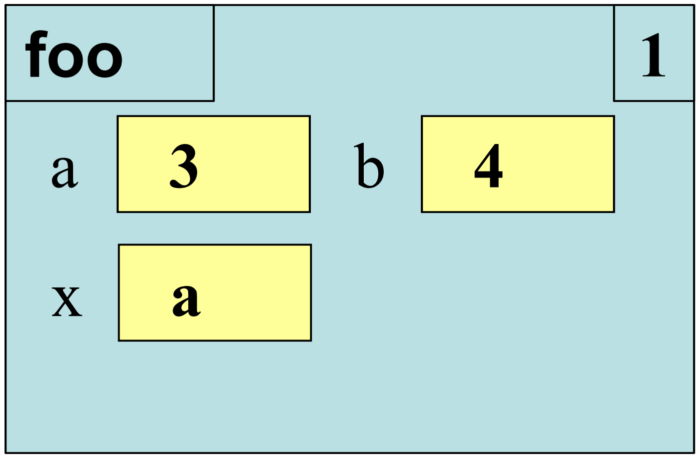
```
def foo(a,b):
    x = a
    y = b
    return x*y+y
```

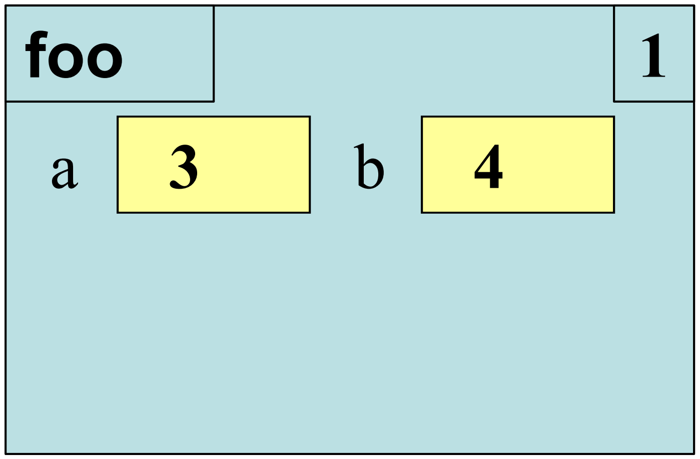
Function Call

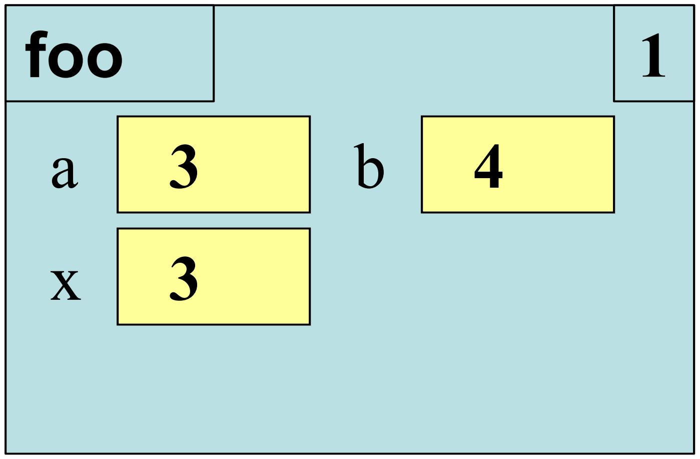
```
>>> foo(3,4)
```

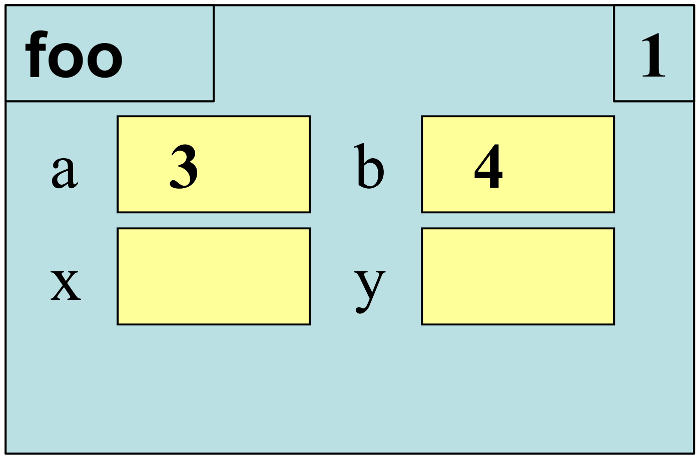
What does the frame look like at the **start**?

Which One is Closest to Your Answer?

A:


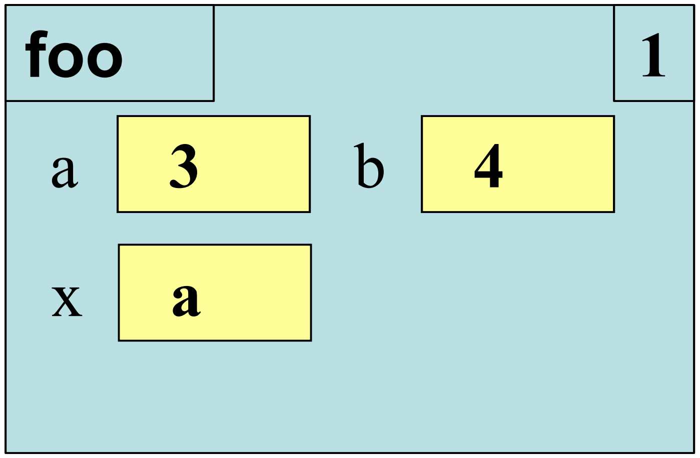
B:


C:


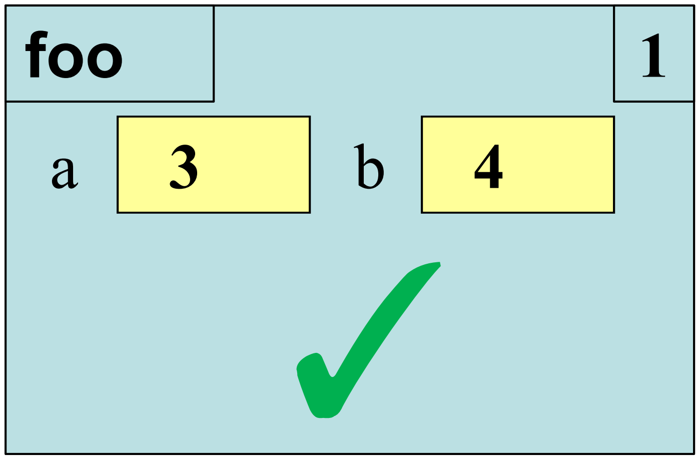
D:


And the answer is…

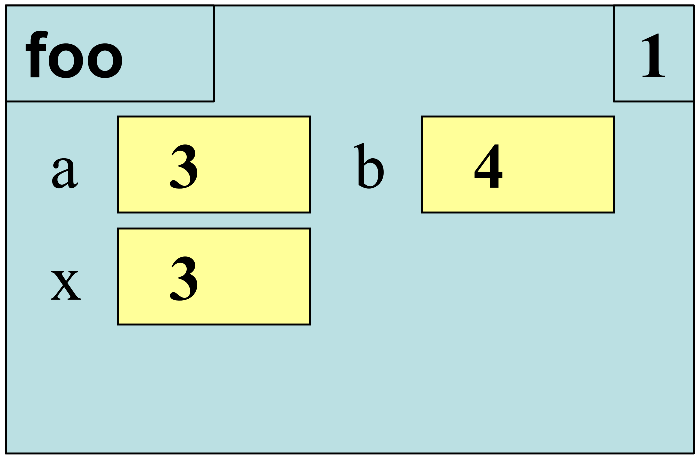
A:



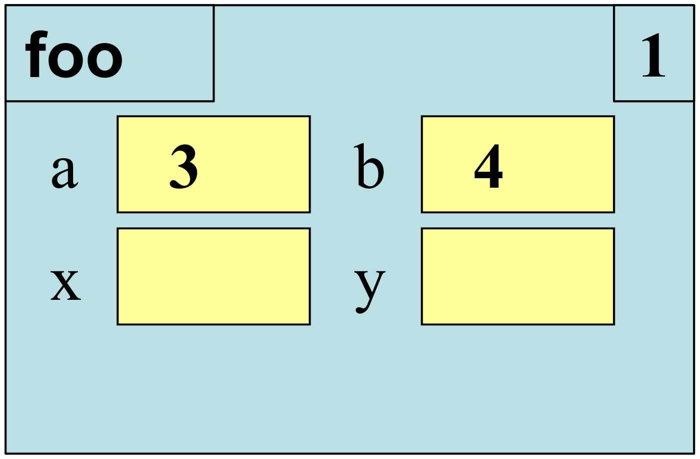
B:



C:



D:



Exercise Time

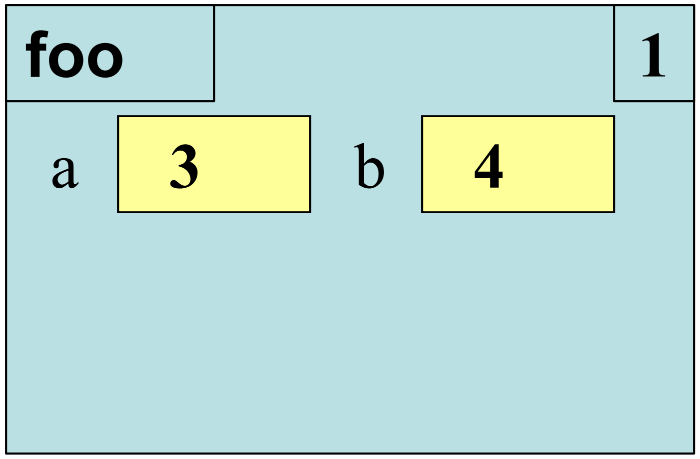
Function Definition

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

Function Call

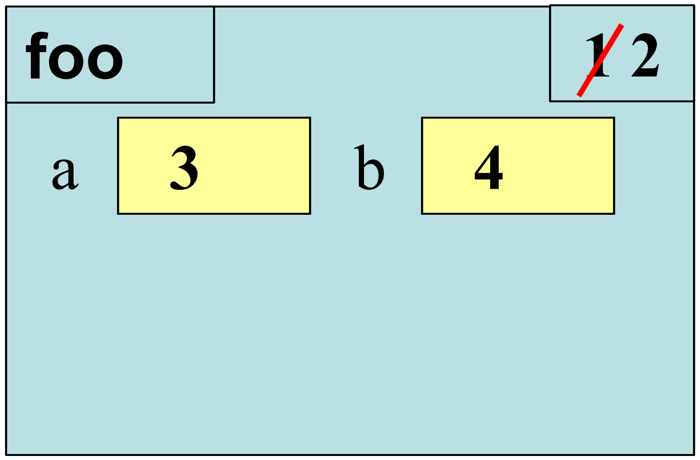
```
>>> foo(3,4)
```

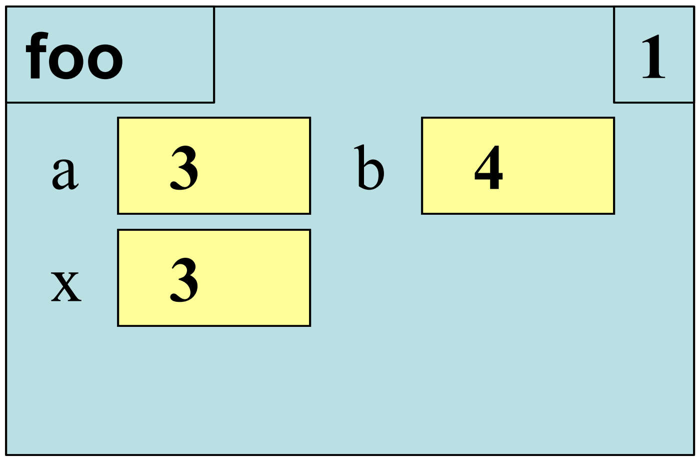
B:

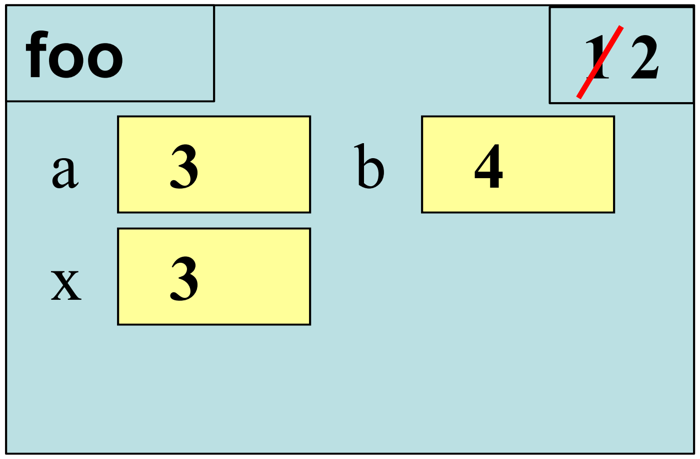


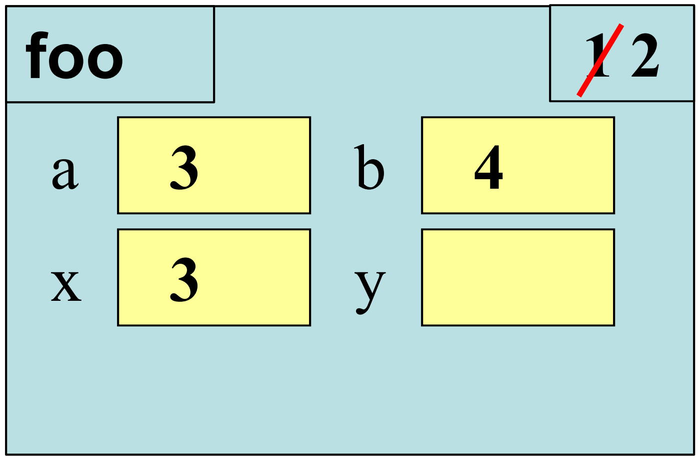
What is the **next step**?

Which One is Closest to Your Answer?

A:


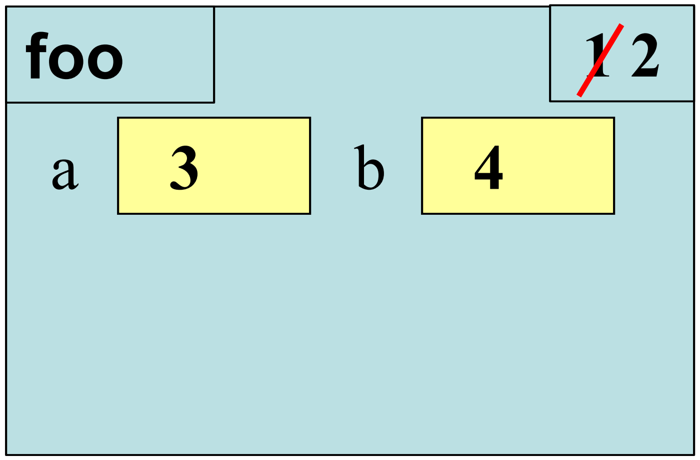
B:


C:


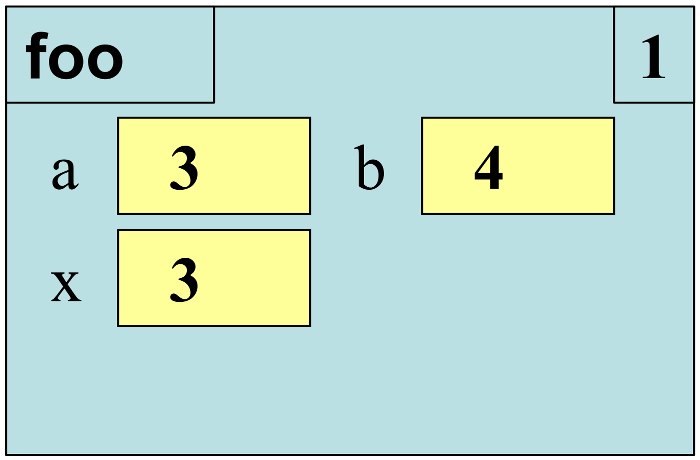
D:


And the answer is…

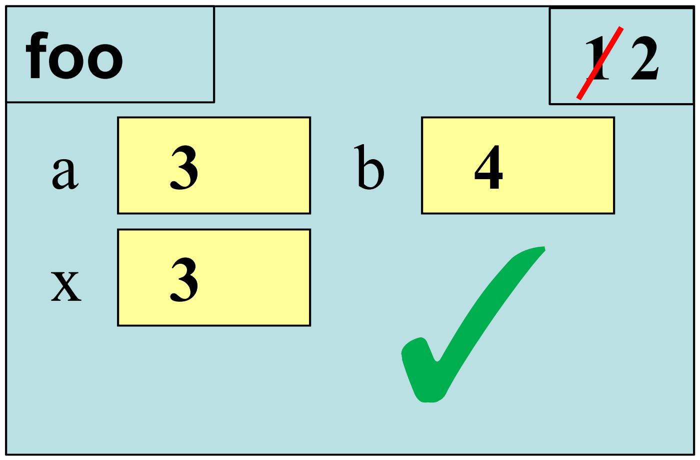
A:



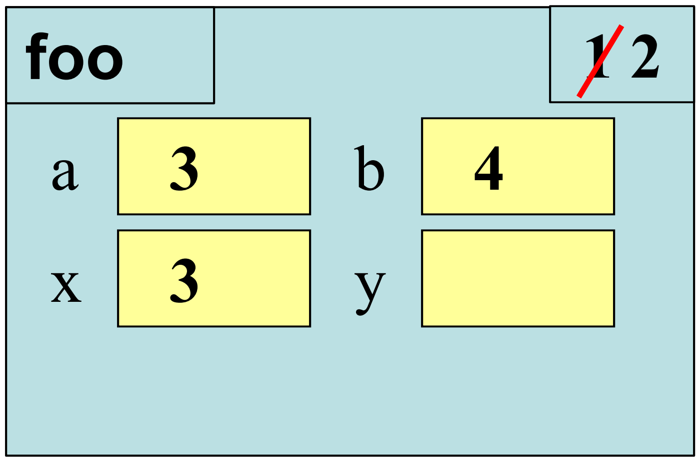
B:



C:



D:



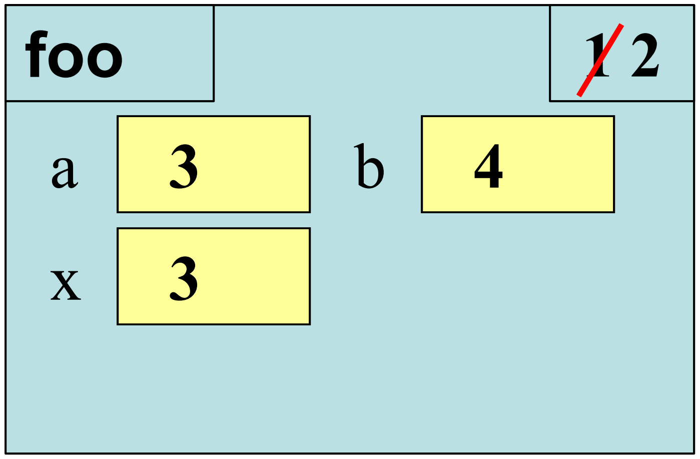
Exercise Time

Function Definition

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

Function Call

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



What is the **next step**?

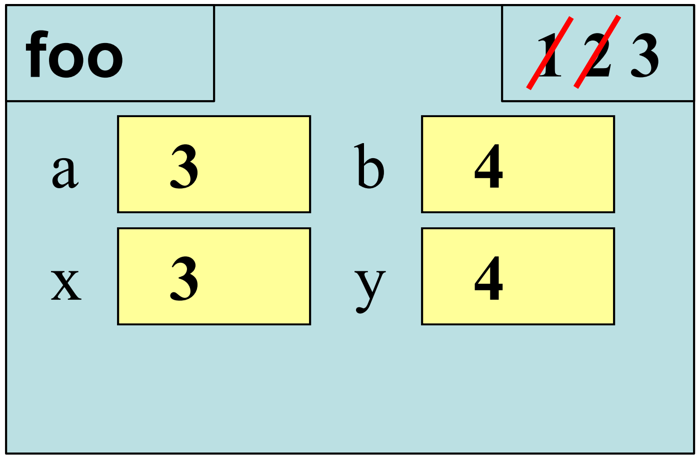
Exercise Time

Function Definition

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

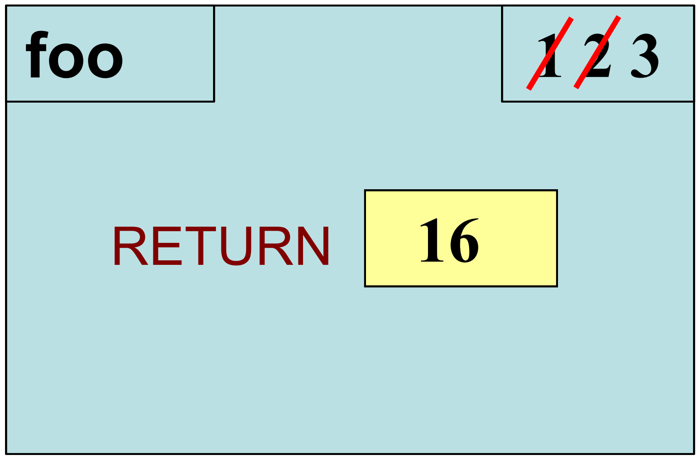
Function Call

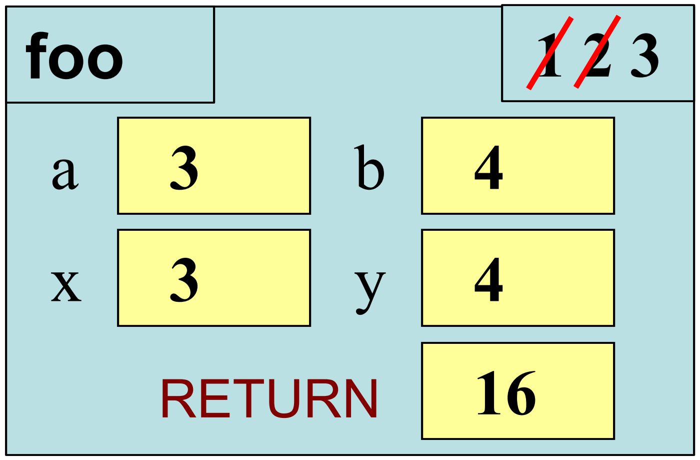
```
>>> foo(3,4)
```

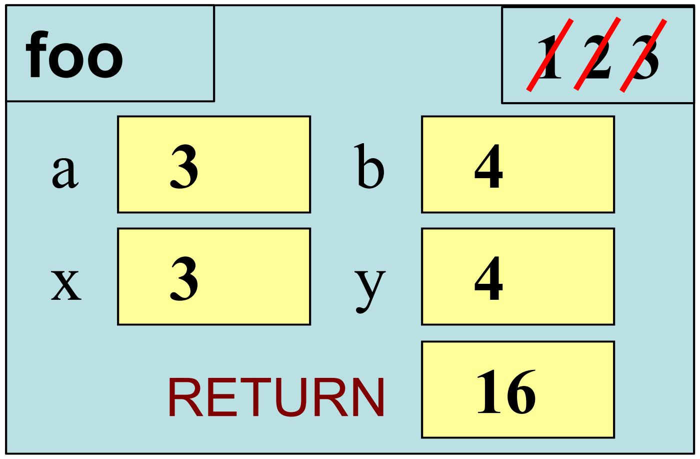


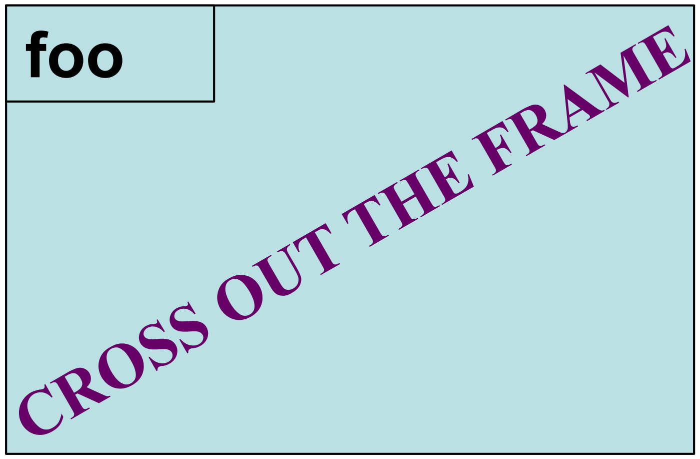
What is the **next step**?

Which One is Closest to Your Answer?

A:


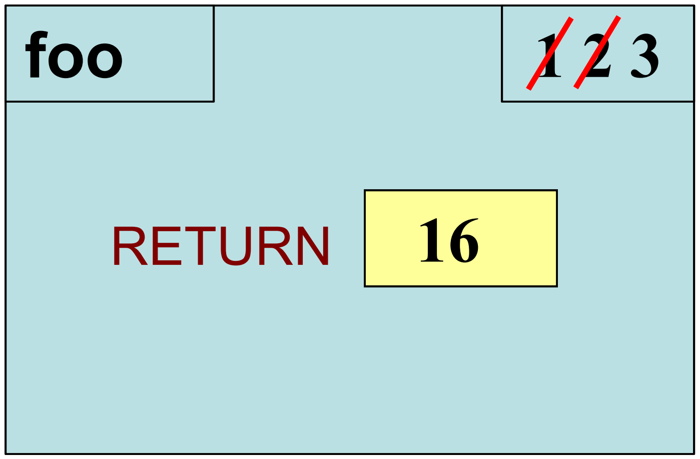
B:


C:


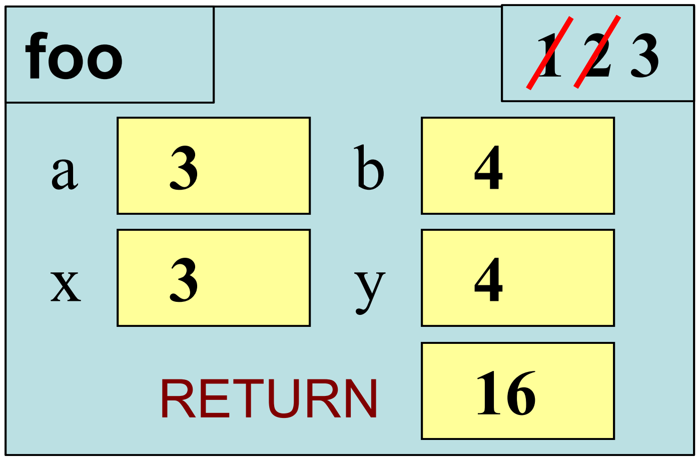
D:


And the answer is…

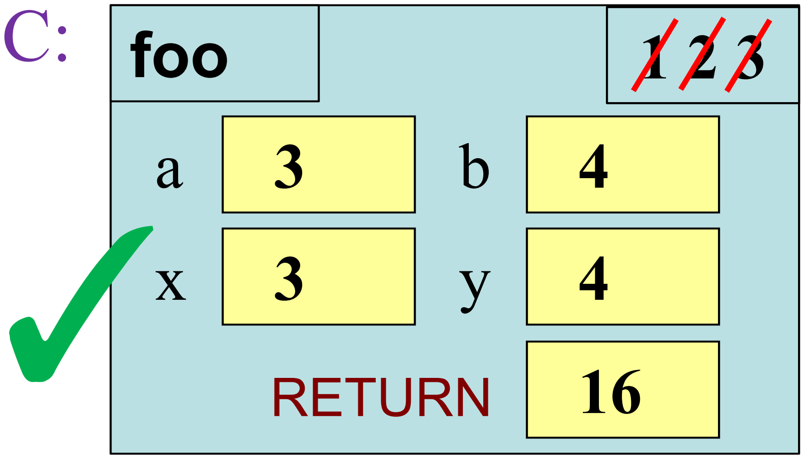
A:



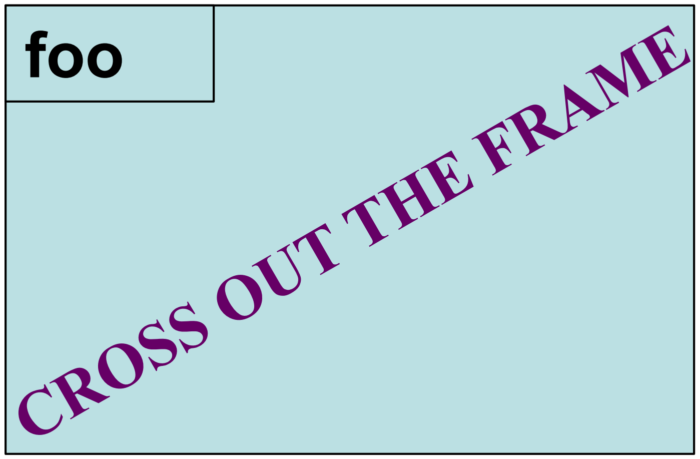
B:



C:



D:



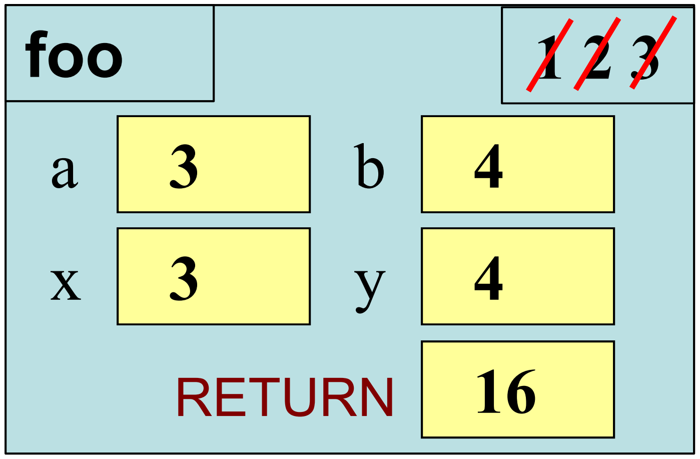
Exercise Time

Function Definition

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

Function Call

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



What is the **next step**?

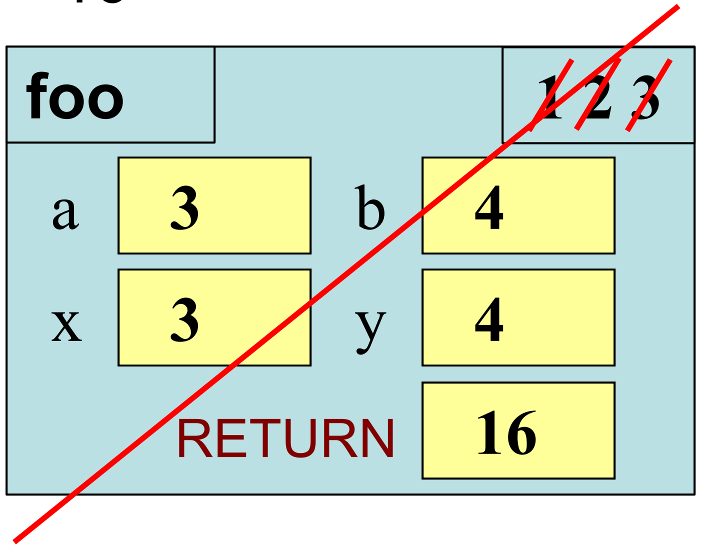
Exercise Time

Function Definition

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

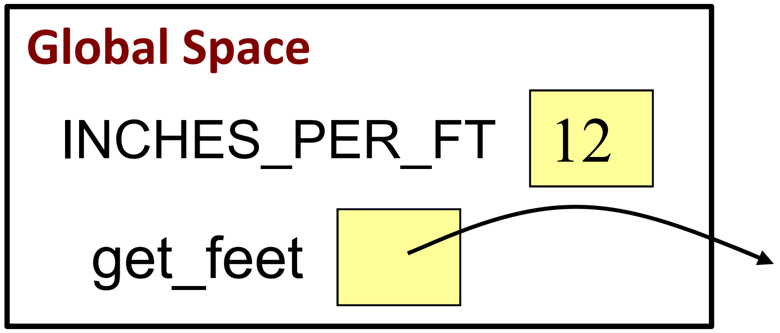
Function Call

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



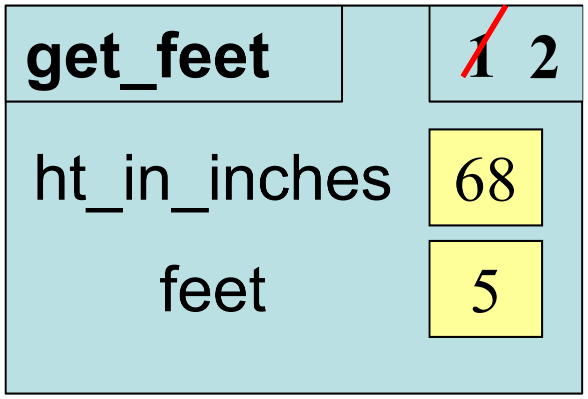
Function Access to Global Space

- Top-most location in memory called global space
- Functions can access anything in that global space



```
INCHES_PER_FT = 12
…
def get_feet(ht_in_inches):
1    feet = ht_in_inches // INCHES_PER_FT
2    return feet

get_feet(68)
```

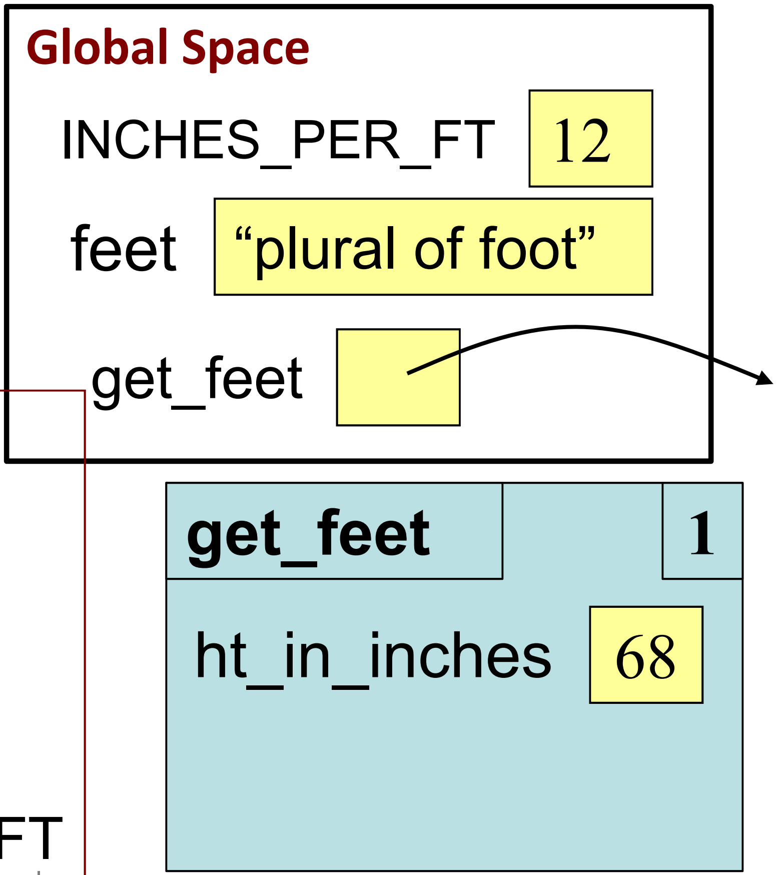


What about this??

- What if you choose a local variable inside a function that happens to also be a global variable?

```
INCHES_PER_FT = 12
feet = "plural of foot"
…
def get_feet(ht_in_inches):
1    feet = ht_in_inches // INCHES_PER_FT
2    return feet

get_feet(68)
```



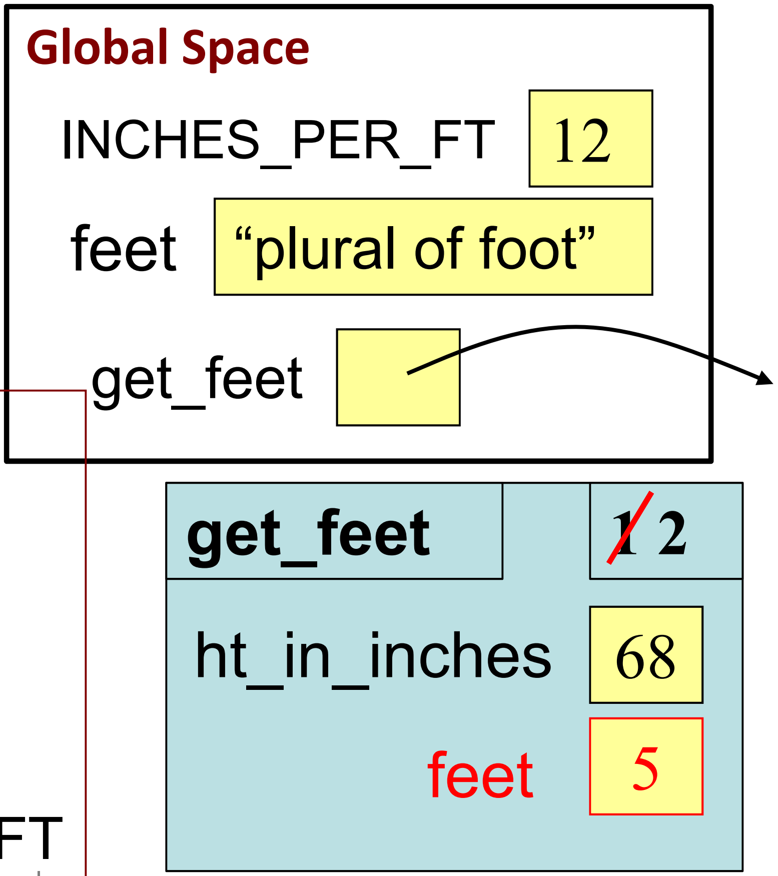
Look, but don't touch!

Can't change global variables

In a function, "assignment to a global" makes a new local variable!

```
INCHES_PER_FT = 12
feet = "plural of foot"
…
def get_feet(ht_in_inches):
1    feet = ht_in_inches // INCHES_PER_FT
2    return feet

get_feet(68)
```



Use “Python Tutor” to help visualize

```
# height2.py

INCHES_PER_FT = 12
feet = "plural of foot"

def get_feet(ht_in_inches):
    """Return ht_in_inches rounded down to nearest feet"""
    feet = ht_in_inches // INCHES_PER_FT
    return feet

get_feet(68)
```

1. Visualize code as is
2. Change code to introduce an error, e.g. misspell **ht_in_inches**. Visualize again.