http://www.cs.cornell.edu/courses/cs1110/2022sp

Lecture 4:
Defining Functions

(Ch. 3.4-3.11)

CS 1110

Introduction to Computing Using Python

Cornell Bowers CIS
Computer Science

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

Lecture Afterthoughts

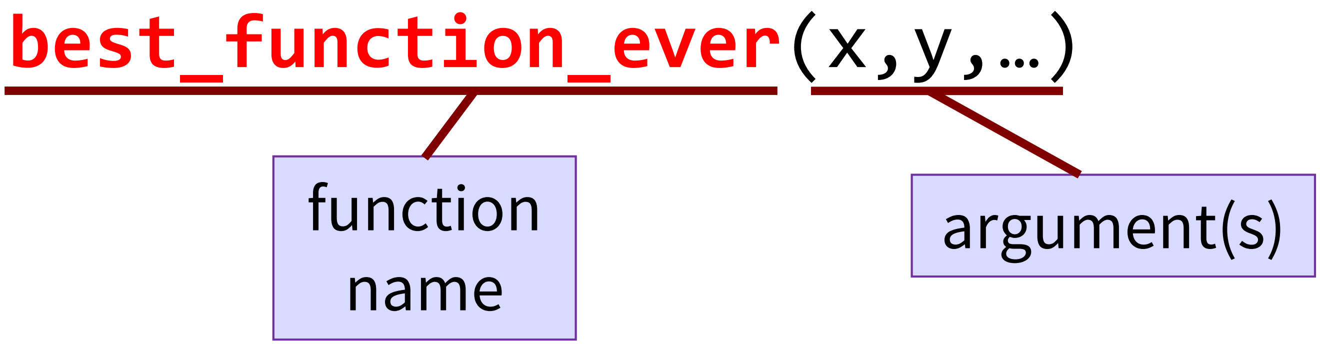
- We added a new slide (#10) to address the question of print vs return. See also this discussion on Ed: https://edstem.org/us/courses/19140/discussion/1084754?comment=2472733
- The lecture concluded with slide 42
- We will cover slides 43-45 at the beginning of the next lecture.
- We strongly suggest you check out the Python Tutor!

Announcements

- Zoom polls not appearing, and not using browser?
  - "a little icon shows up on the bottom … sometimes you have to click it to see the poll."(Thanks, CS1110 student for the tip!)

From Last Time: Function Calls

- Function calls have the form:



```
best_function_ever(x,y,…)
```

function name

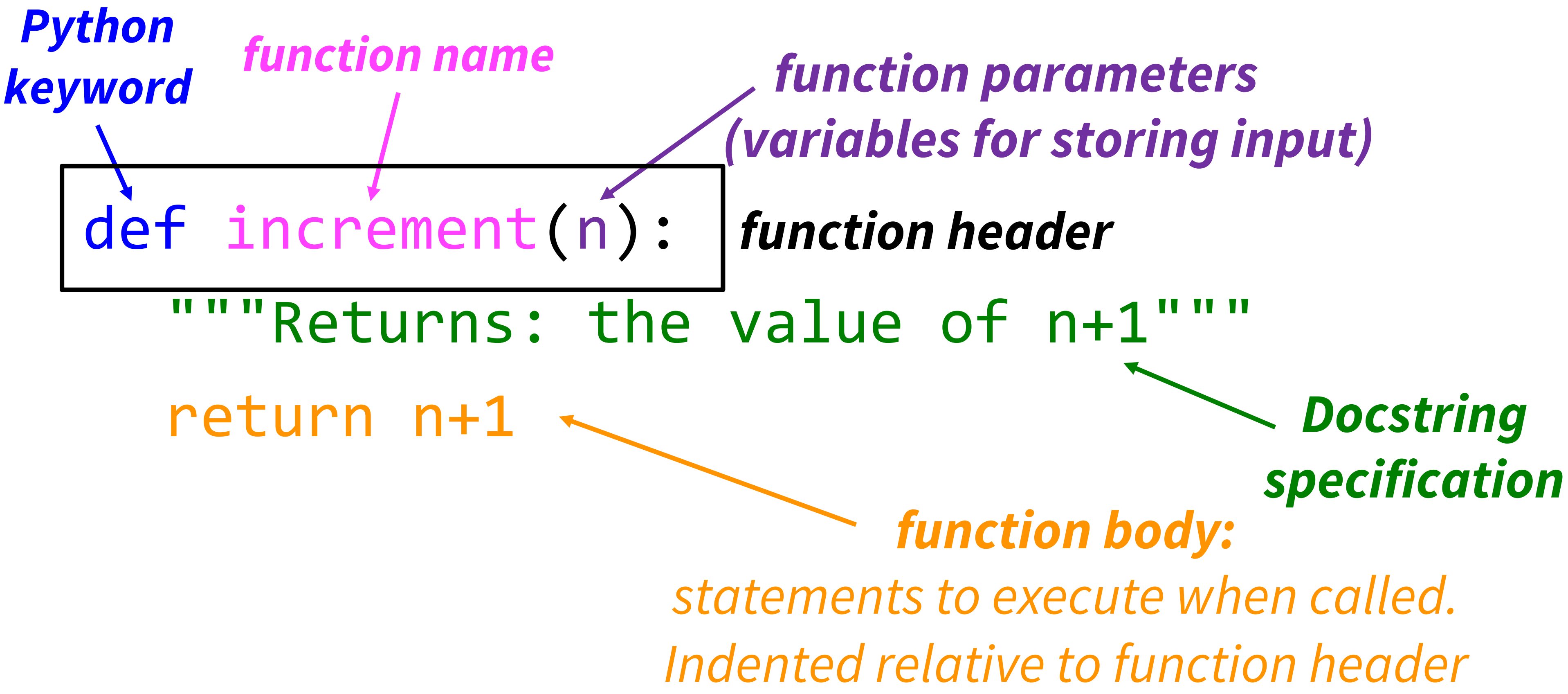
argument(s)

- Arguments: values given as inputs
  - Separated by commas
  - Can be any expression

A function might have 0, 1, … or many arguments

Let’s define our own functions!

Anatomy of a Function Definition



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

Python keyword

function name

function parameters (variables for storing input)

function header

Docstring specification

function body:

statements to execute when called. Indented relative to function header

The **`return`** Statement

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

Organization of a Module

```python
# simple_math.py


def increment(n):
    return n+1



increment(2)
```

simple_math.py

- Function definition goes before any code that calls that function
- There can be multiple function definitions
- Can organize function definitions in any order

Function Definitions vs. Calls

```
# simple_math.py


def increment(n):
    return n+1



increment(2)
```

simple_math.py

Function definition

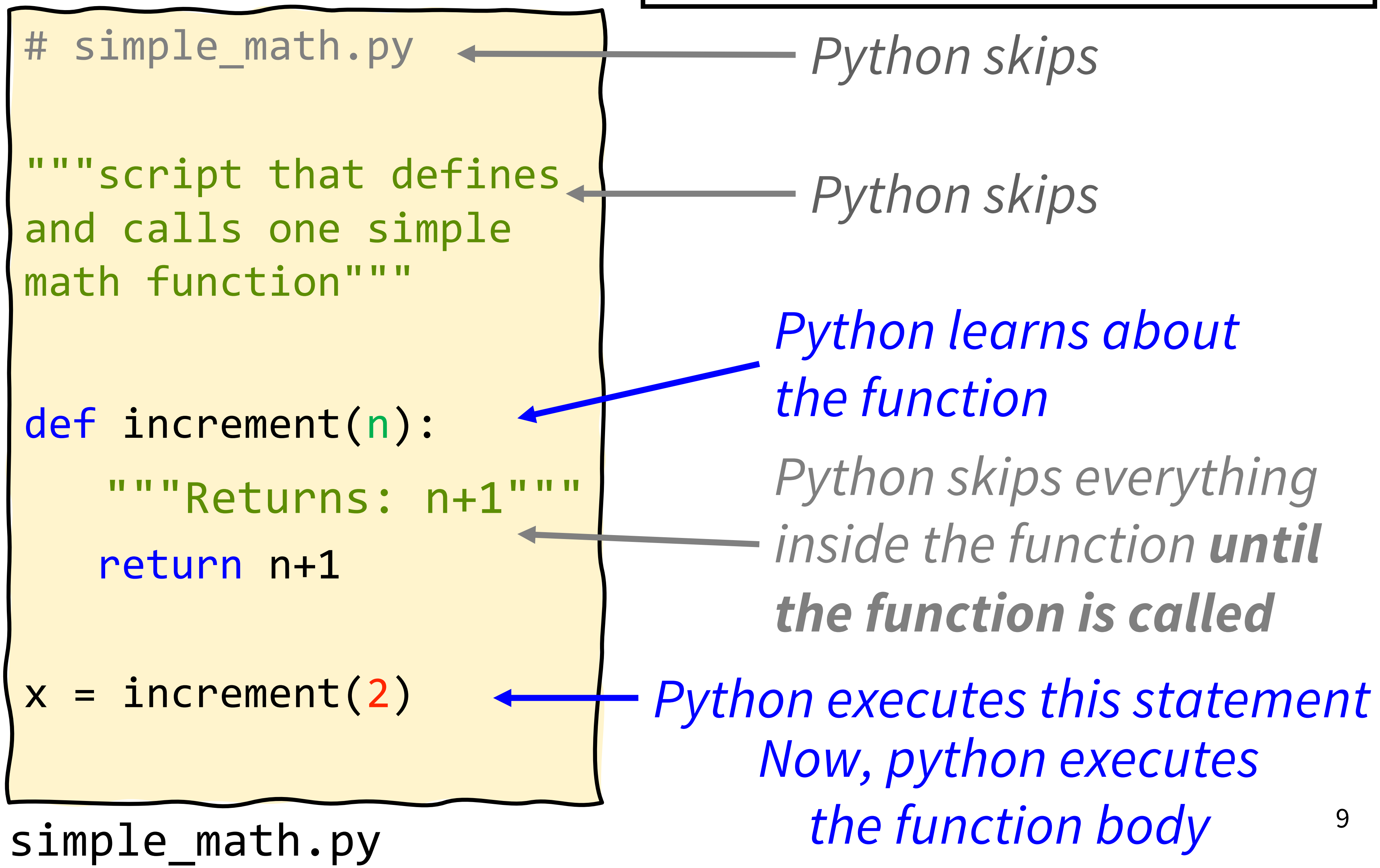
- Defines what function will do
- Declaration of parameters (n in this case)
- **Parameter:** variable where input to function is stored

Function call

- Command to do the function
- Argument to assign to function parameter (Argument **2** to be assigned to parameter **n** in this case)
- **Argument:** an input value to assign to the function parameter when it is called

Executing the script `simple_math.py`

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



```
# simple_math.py

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


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

x = increment(2)
```

simple_math.py

Python skips

Python skips

Python learns about the function

Python skips everything inside the function ***until the function is called***

Python executes this statement

Now, python executes the function body

return vs. print

https://edstem.org/us/courses/19140/discussion/1084754?comment=2472733

```
# simple_math.py

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

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

x = increment(2)
```

simple_math.py

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

Notice that this script does not print anything!

*The function **returns** the value (it gets saved in x) but does not print it.*

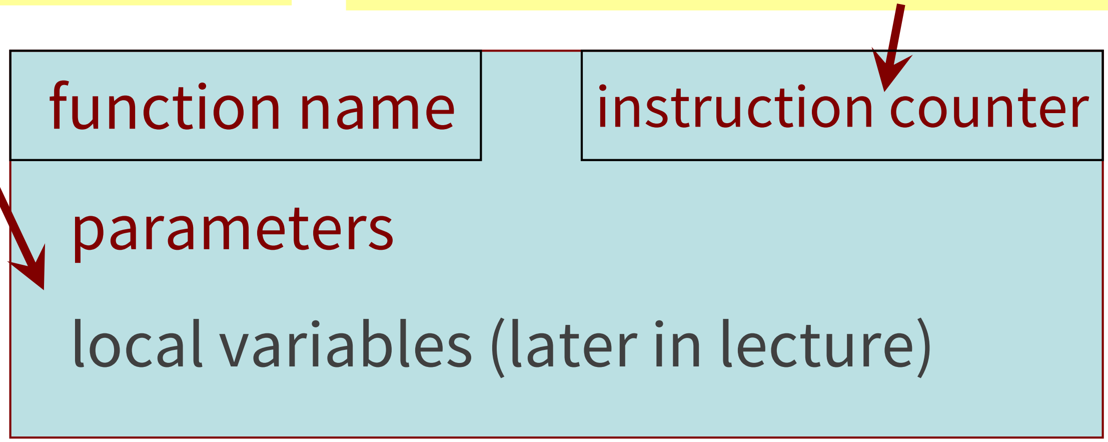
If you want the function to also print to the screen, it needs a print statement.

Understanding How Functions Work

- We draw pictures to show what is in memory
- **Call Frame:** representation of function call

Draw parameters as variables (named boxes)

- Line number of the **next** statement in the function body to execute
- Starts with 1st statement in function body



function name

instruction counter

parameters

local variables (later in lecture)

Not just a pretty picture!

The information in this picture depicts *exactly* what is stored in memory on your computer.

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)
```

```
# height.py

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

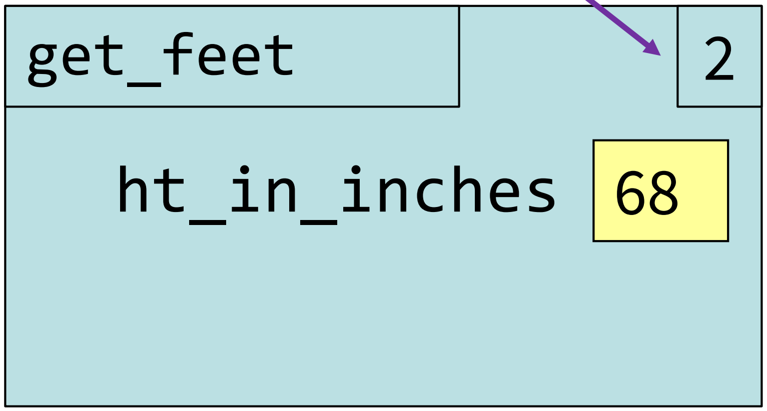
height.py

Example: `get_feet(68)` (slide 1)

```
>>> import height
>>> height.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


next line to execute

```
# height.py

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

height.py

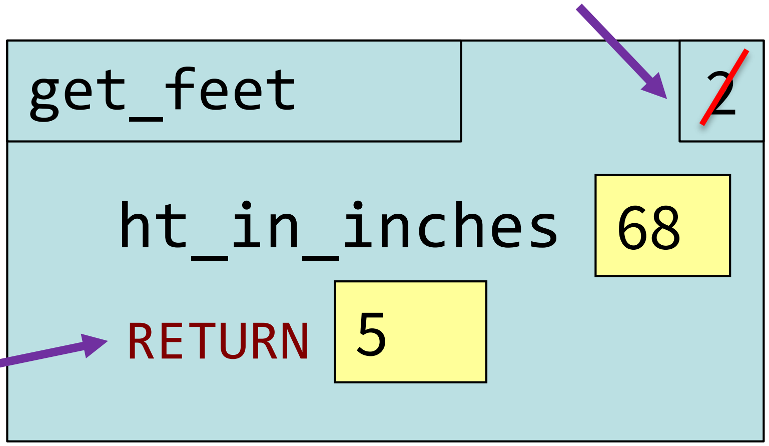
Example: `get_feet(68)` (slide 2)

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

PHASE 2:

Execute function body

The return terminates; no next line to execute



get_feet ~~2~~

ht_in_inches 68

RETURN 5

Return statement creates a special variable for result

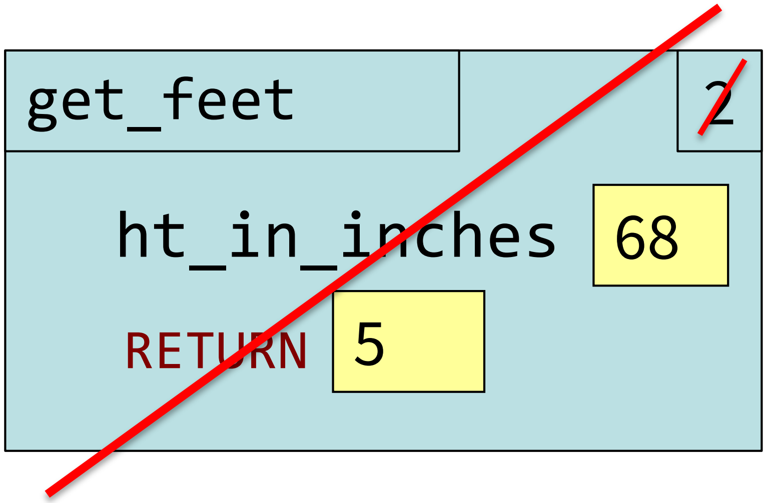
```
# height.py

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

height.py

Example: get_feet(68) (slide 3)

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

Python interactive mode
evaluates the expression and reports



get_feet ~~2~~

ht_in_inches 68

RETURN 5

PHASE 3: Delete (cross out) call frame

```
  # height.py

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

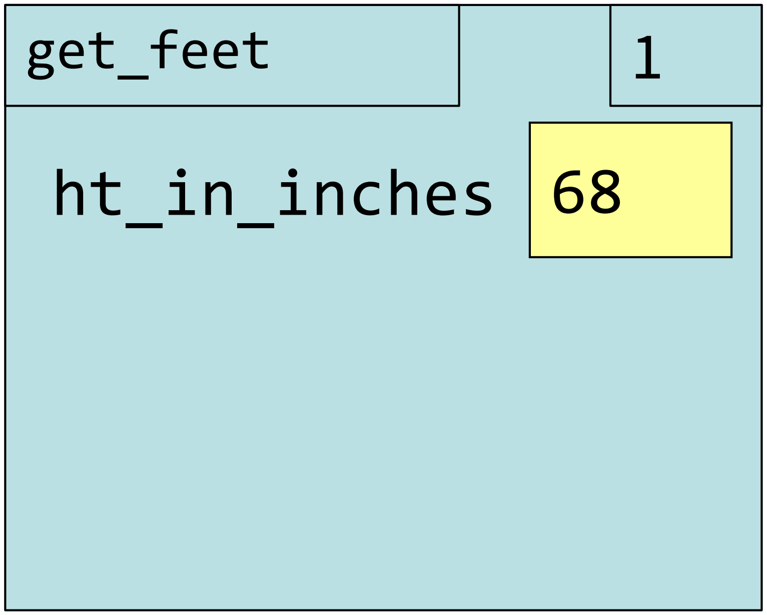
height.py

Local Variables (1)

Call frames can contain "local" variables

- A variable created in the function

```
>>> import height2
>>> height2.get_feet(68)
```



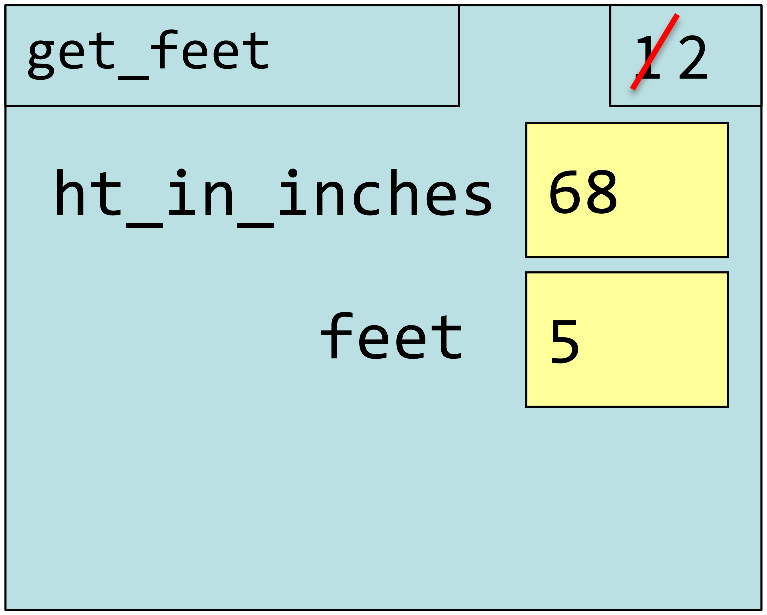
```
# height2.py

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

height2.py

Local Variables (2)

Call frames can contain “local” variables

```
>>> import height2
>>> height2.get_feet(68)
```



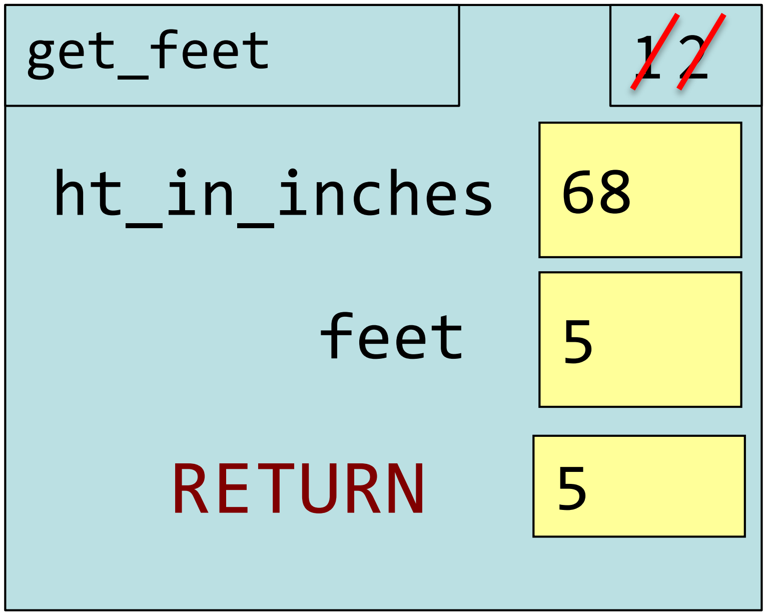
```
  # height2.py

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

height2.py

Local Variables (3)

Call frames can contain "local" variables

```
>>> import height2
>>> height2.get_feet(68)
```



```
  # height2.py

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

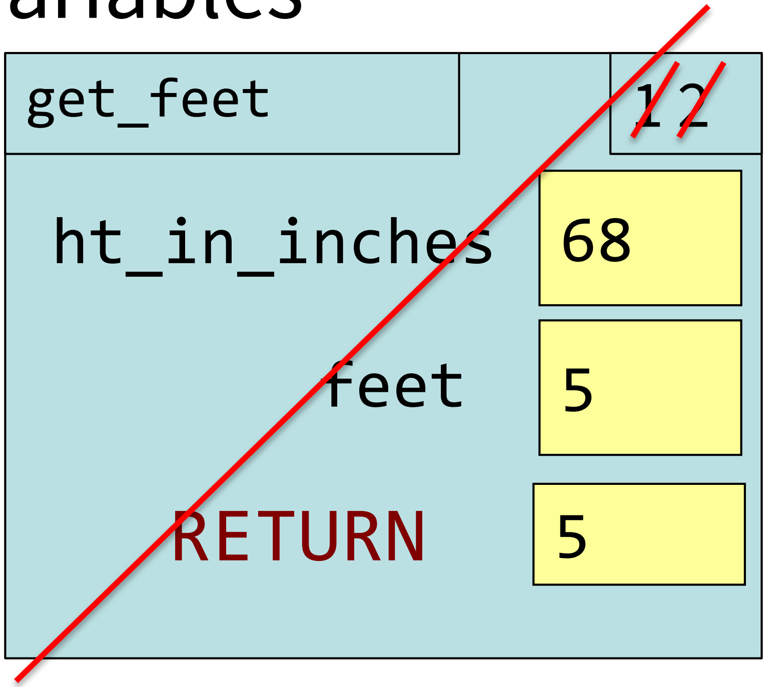
height2.py

Local Variables (4)

Call frames can contain "local" variables

```
>>> import height2
>>> height2.get_feet(68)
5
>>>
```

Python interactive mode
evaluates the expression and reports



get_feet ~~1~~ ~~2~~

ht_in_inches 68

feet 5

RETURN 5

```
# height2.py

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

height2.py

Variables are gone!
This function is over.

Exercise #1

Function Definition

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

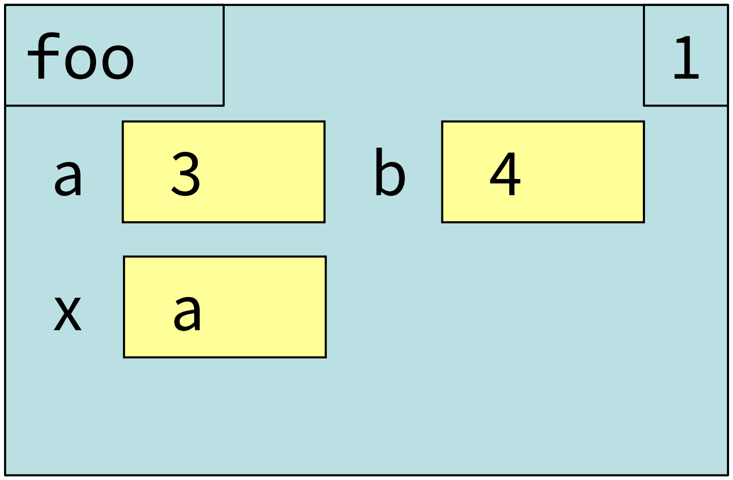
Function Call

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

What does the frame look like at the start?

Which One is Closest to Your Answer?

A:

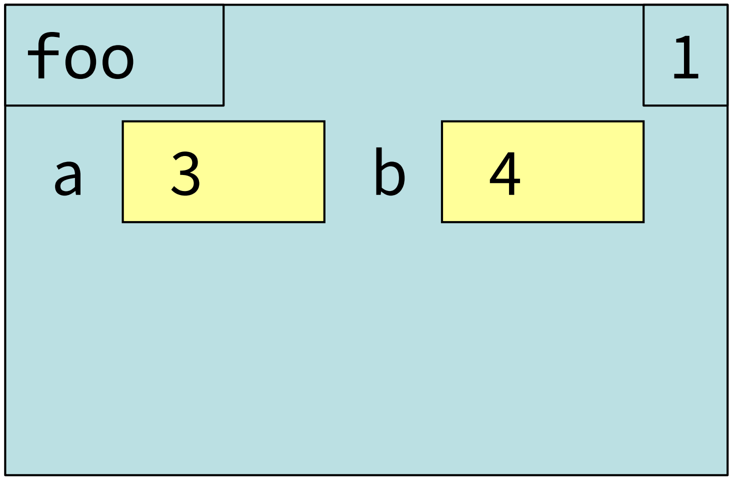


foo 1

a 3 b 4

x a

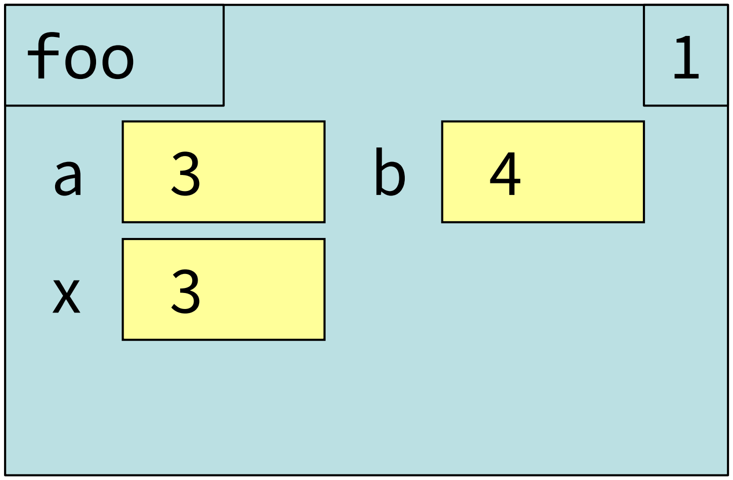
B:



foo 1

a 3 b 4

C:

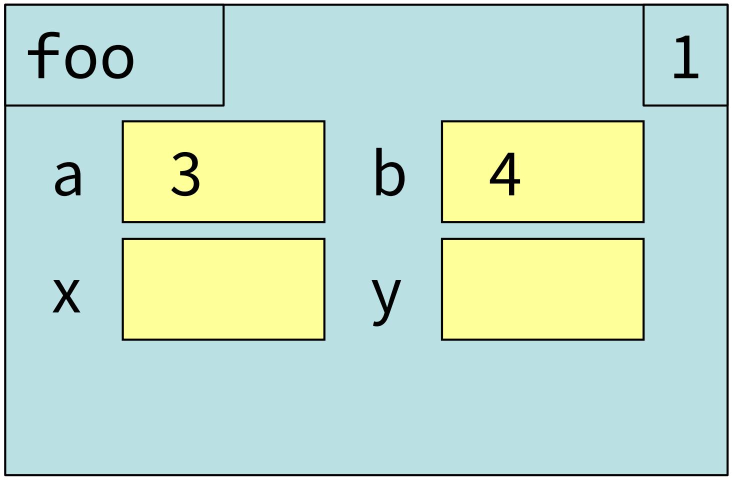


foo 1

a 3 b 4

x 3

D:



foo 1

a 3 b 4

x y

Exercise #2

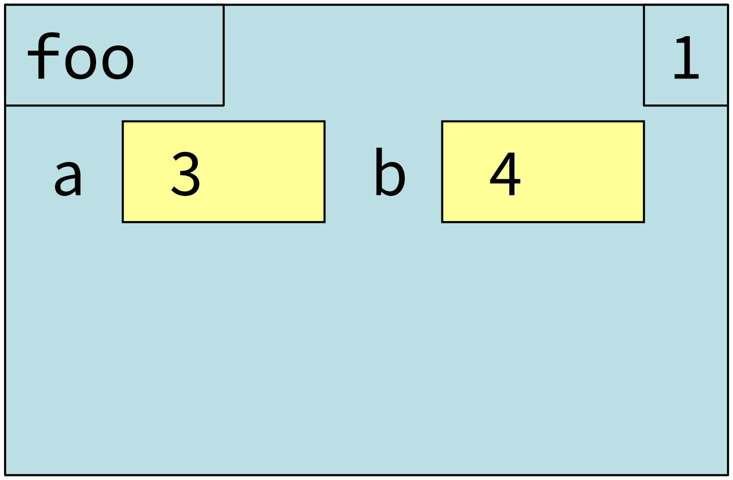
Function Definition

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

Function Call

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

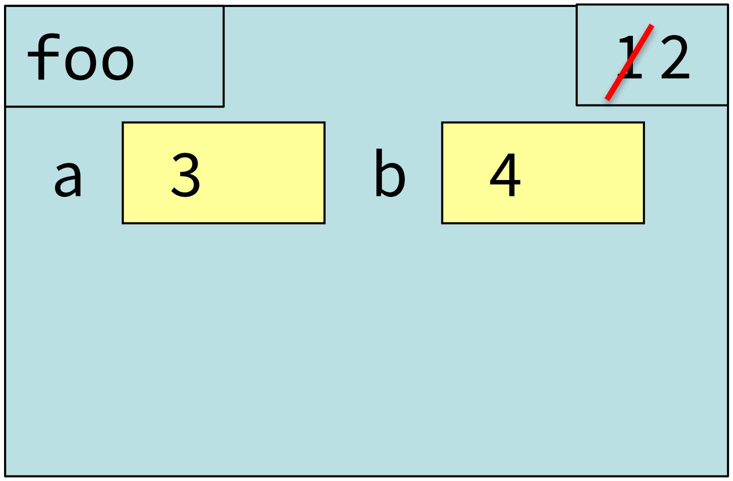
B:



What is the next step?

Which One is Closest to Your Answer?

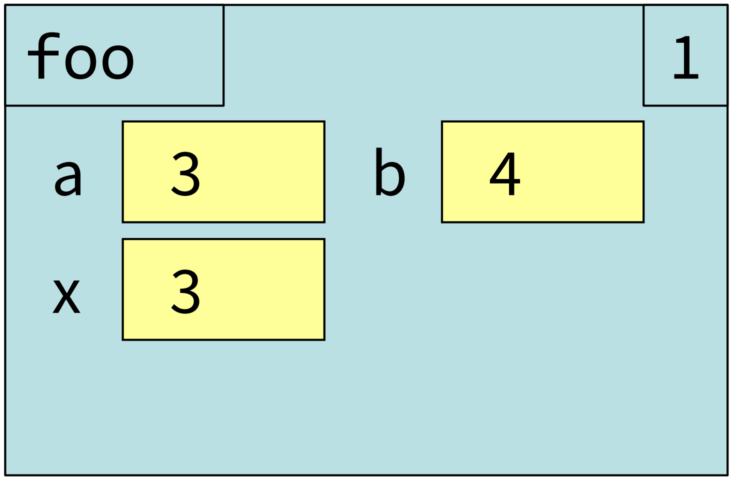
A:



foo ~~1~~ 2

a 3 b 4

B:

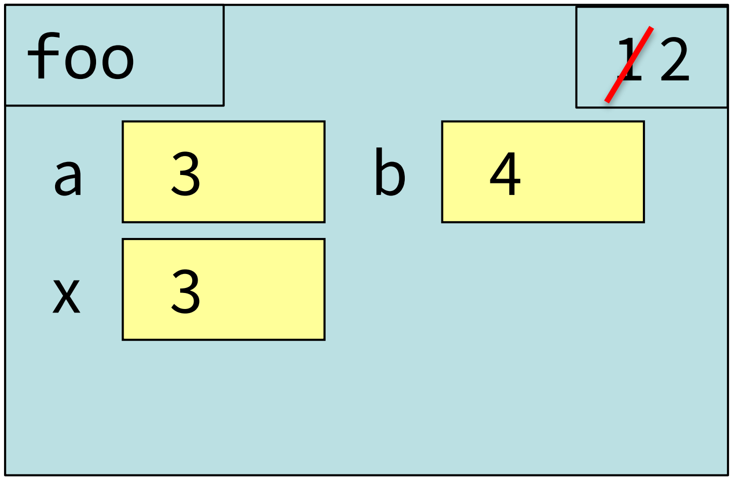


foo 1

a 3 b 4

x 3

C:

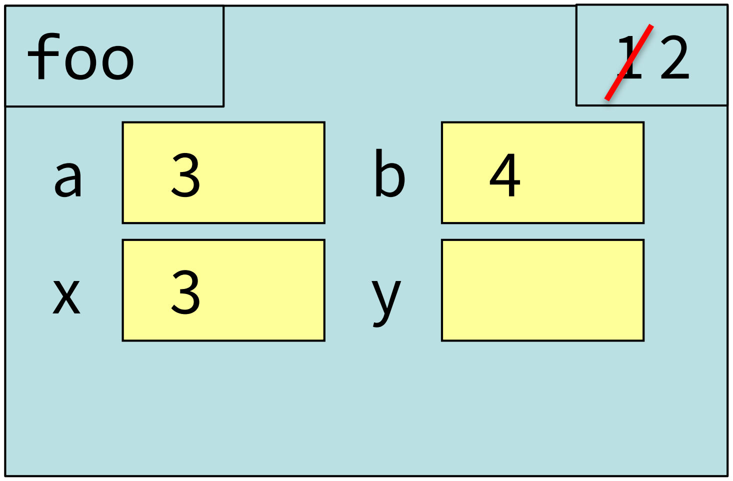


foo ~~1~~ 2

a 3 b 4

x 3

D:



foo ~~1~~ 2

a 3 b 4

x 3 y

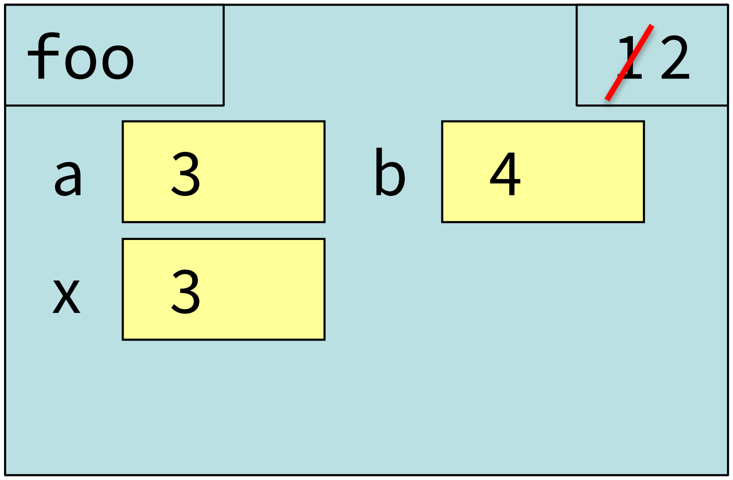
Exercise Time *(no poll, just discuss)*

Function Definition

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

Function Call

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



foo ~~1~~ 2

a 3 b 4

x 3

What is the next step?

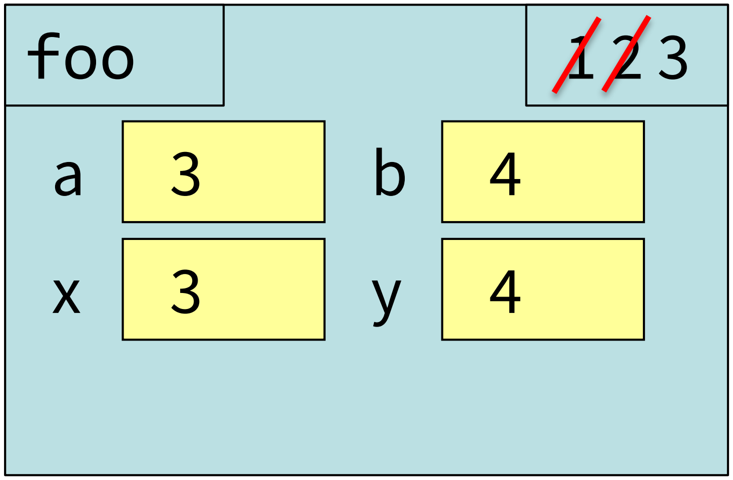
Exercise #3

Function Definition

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

Function Call

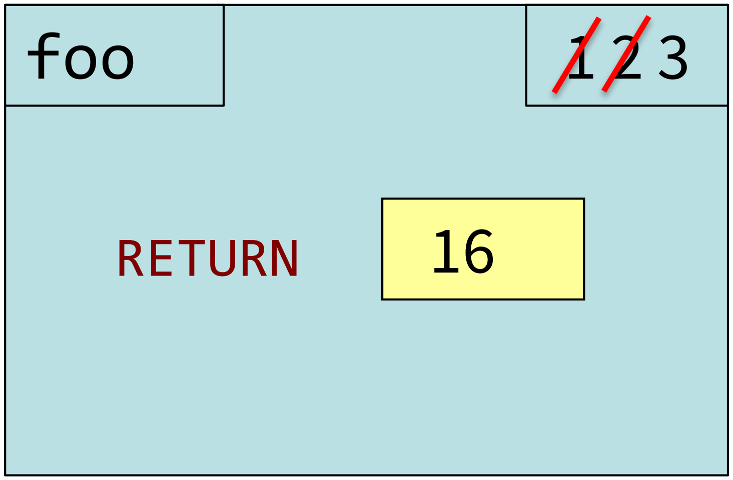
```
>>> foo(3,4)
```



What is the **next step**?

Which One is Closest to Your Answer?

A:

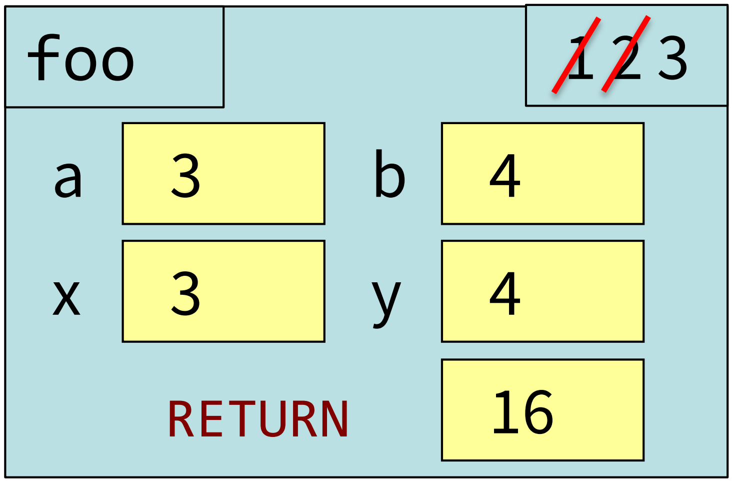


foo

~~1~~ ~~2~~ 3

RETURN 16

B:



foo

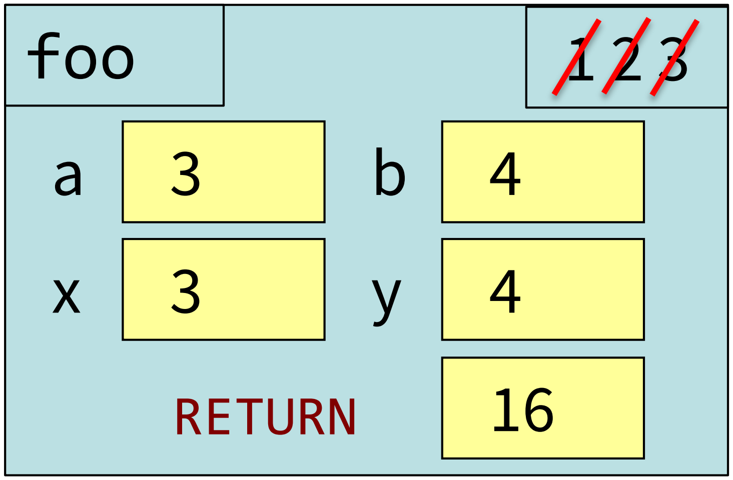
~~1~~ ~~2~~ 3

a 3 b 4

x 3 y 4

RETURN 16

C:



foo

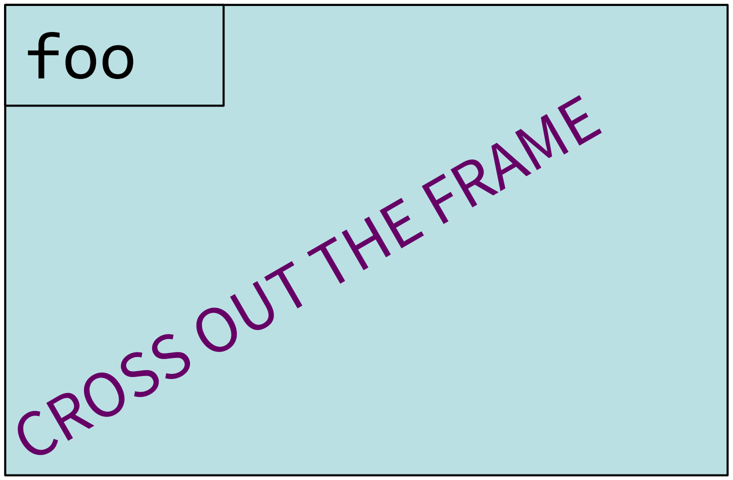
~~1~~ ~~2~~ ~~3~~

a 3 b 4

x 3 y 4

RETURN 16

D:



foo

CROSS OUT THE FRAME

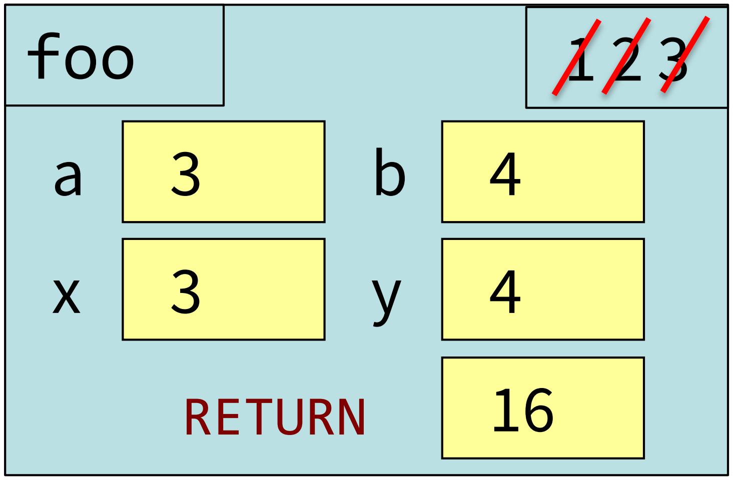
Exercise Time *(no poll, just discuss)*

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?

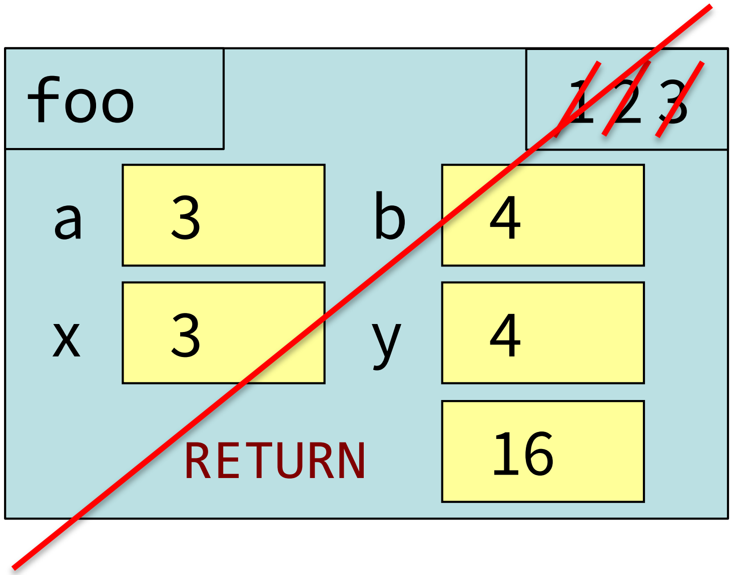
Exercise Time

Function Definition

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

Function Call

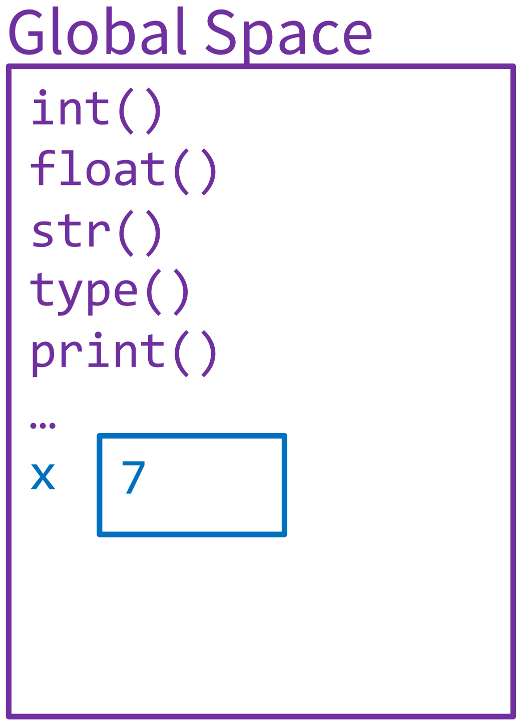
```
>>> foo(3,4)
>>> 16
```



Global Space

= the purple box we previously labeled
“What Python can access directly”

- Top-most location in memory
- Variables in Global Space called Global Variables
- Functions can access anything global space (see next slides)



Global Space

```
int()
float()
str()
type()
print()
…
x    7
```

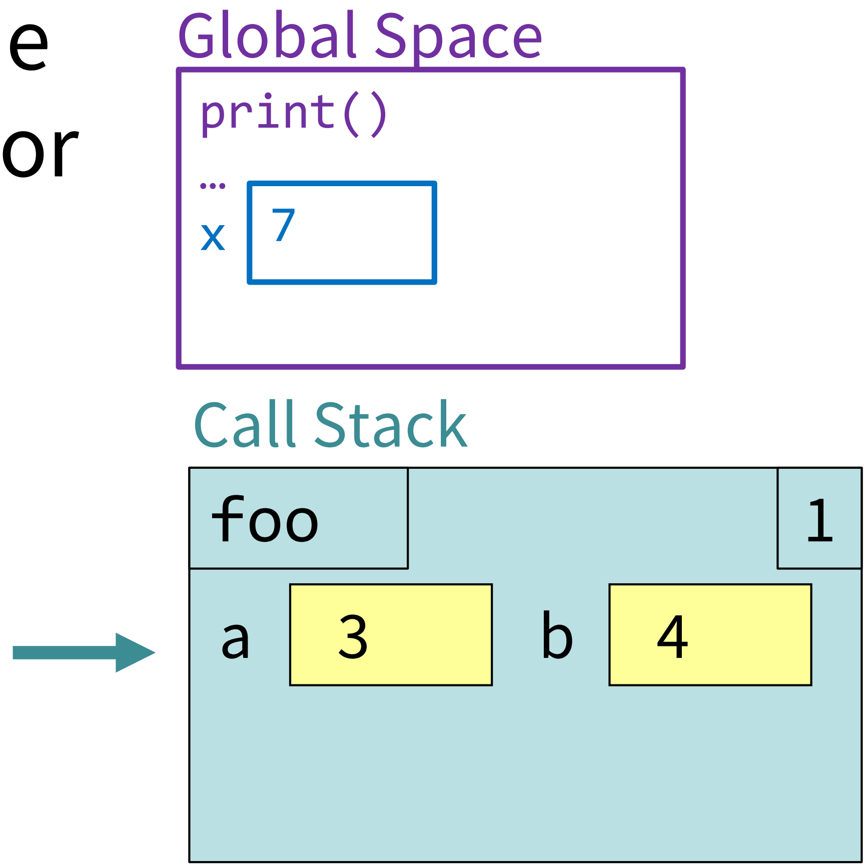
```
C:\> python
>>> x = 7
>>>
```

Call Stack

= the place in memory where the Call Frames live

Functions can only access the variables in their Call Frame or the Global Space.

This is the Call Frame for the function **foo***. It is created in response to a function call and lives on the Call Stack, distinct from the Global Space.*



Global Space

print()

…

x 7

Call Stack

foo 1

a 3 b 4

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

Function Access to Global Space (1)

```
# height3.py

INCHES_PER_FT = 12
def get_feet(ht_in_inches):
    feet = ht_in_inches // INCHES_PER_FT
    return feet

answer = get_feet(68)
print(answer)
```

Global Space

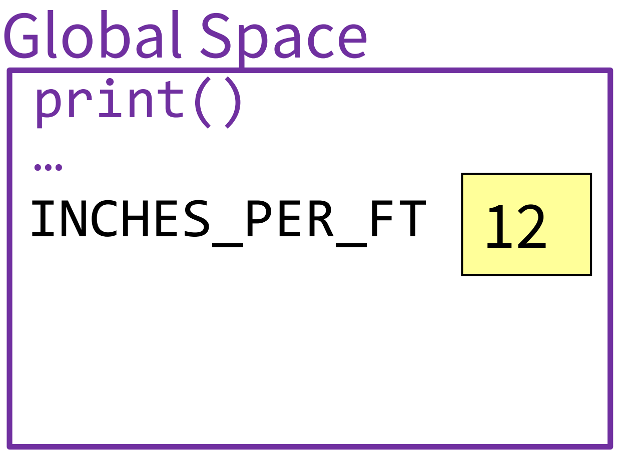
```
print()
…
```

```
C:\> python height3.py
```

Python just started.
It has all the built-in functions.
It hasn’t read any of the module yet.

Function Access to Global Space (2)

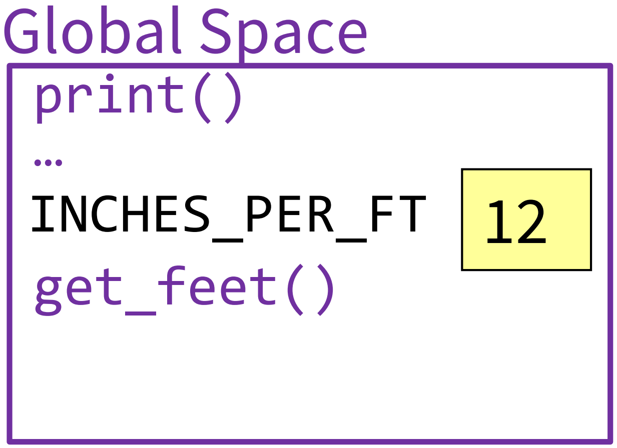
```
  # height3.py

1 INCHES_PER_FT = 12
2 def get_feet(ht_in_inches):
3     feet = ht_in_inches // INCHES_PER_FT
4     return feet

5 answer = get_feet(68)
6 print(answer)
```



Python just read line 1 of the module. A variable has been added to the Global Space.

Function Access to Global Space (3)

```
  # height3.py

1 INCHES_PER_FT = 12
2 def get_feet(ht_in_inches):
3     feet = ht_in_inches // INCHES_PER_FT
4     return feet

5 answer = get_feet(68)
6 print(answer)
```



Global Space

print()

…

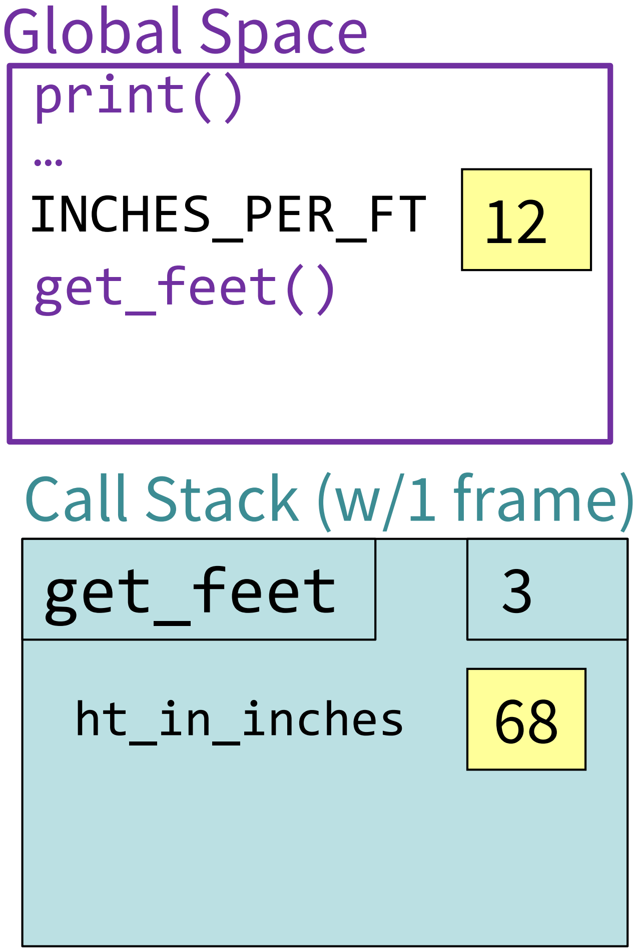
INCHES_PER_FT 12

get_feet()

Python just read line 2 of the module.
A new function has been added to the Global Space.
Note: python has not yet looked inside the function.

Function Access to Global Space (4)

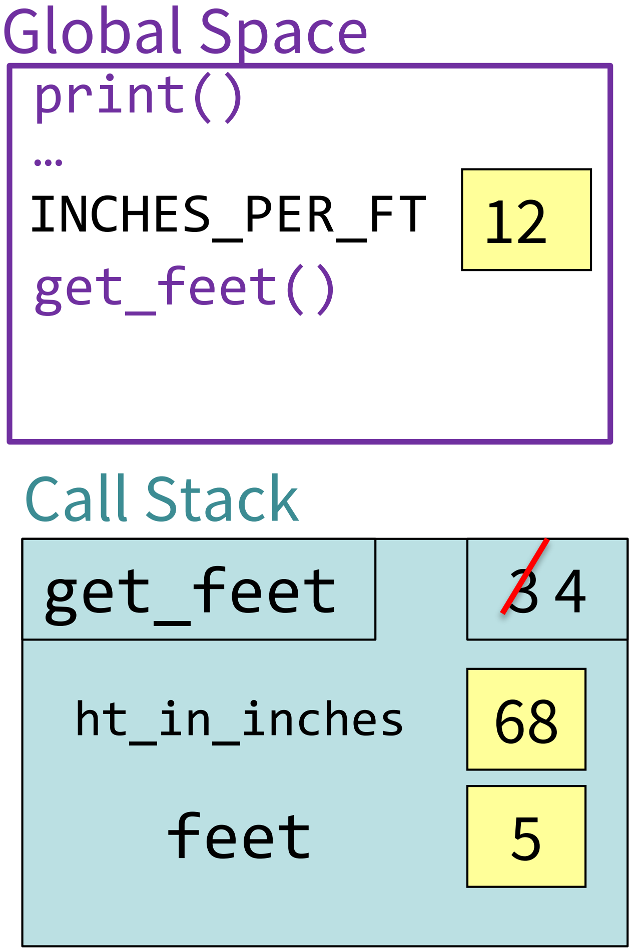
```
  # height3.py

1 INCHES_PER_FT = 12
2 def get_feet(ht_in_inches):
3     feet = ht_in_inches // INCHES_PER_FT
4     return feet

5 answer = get_feet(68)
6 print(answer)
```



To execute the assignment statement on line 5, Python needs to evaluate the RHS. Python creates a call frame for the function, which lives on the Call Stack.

Function Access to Global Space (5)

```
  # height3.py

1 INCHES_PER_FT = 12
2 def get_feet(ht_in_inches):
3     feet = ht_in_inches // INCHES_PER_FT
4     return feet

5 answer = get_feet(68)
6 print(answer)
```



Global Space

print()

…

INCHES_PER_FT 12

get_feet()

Call Stack

get_feet ~~3~~ 4

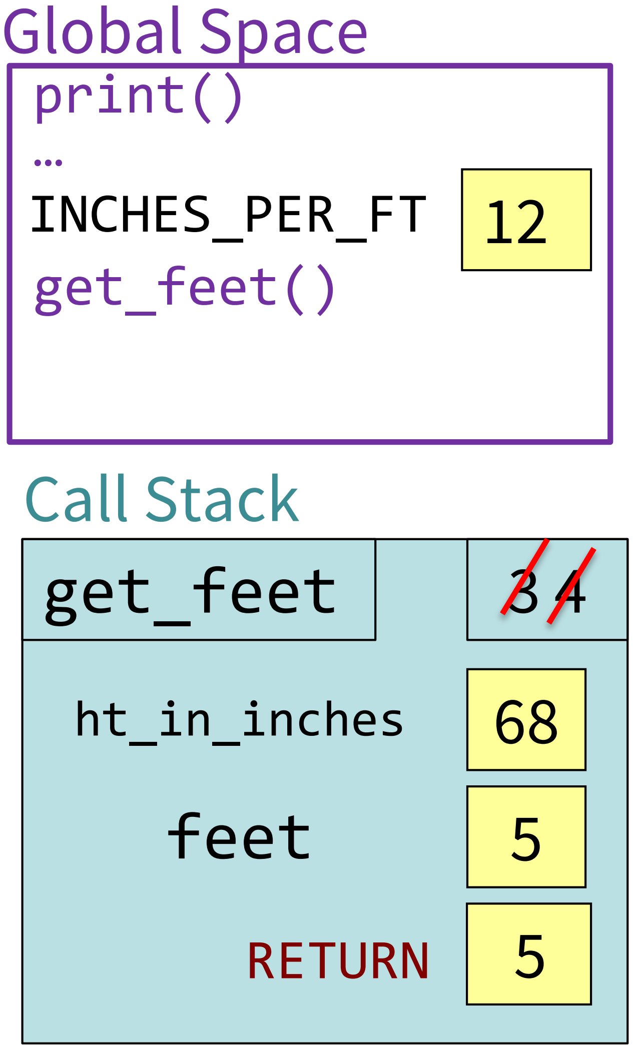
ht_in_inches 68

feet 5

Python has just executed line 3.
A new local variable feet has been created inside get_feet's Call Frame.

Function Access to Global Space (6)

```
  # height3.py

1 INCHES_PER_FT = 12
2 def get_feet(ht_in_inches):
3     feet = ht_in_inches // INCHES_PER_FT
4     return feet

5 answer = get_feet(68)
6 print(answer)
```



Global Space

print()

…

INCHES_PER_FT 12

get_feet()

Call Stack

get_feet ~~3~~ ~~4~~

ht_in_inches 68

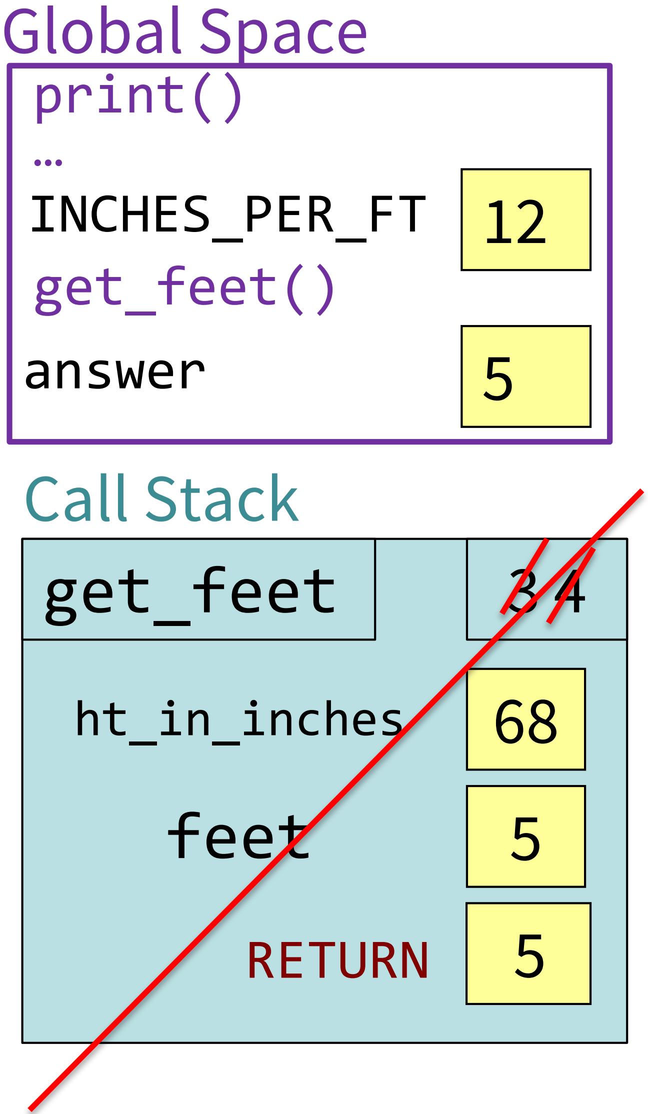
feet 5

RETURN 5

Python has just executed line 4.
A return value has been created.

Function Access to Global Space (7)

```
  # height3.py

1 INCHES_PER_FT = 12
2 def get_feet(ht_in_inches):
3     feet = ht_in_inches // INCHES_PER_FT
4     return feet

5 answer = get_feet(68)
6 print(answer)
```



Global Space

print()

…

INCHES_PER_FT 12

get_feet()

answer 5

Call Stack

get_feet ~~3~~ ~~4~~

ht_in_inches 68

feet 5

RETURN 5

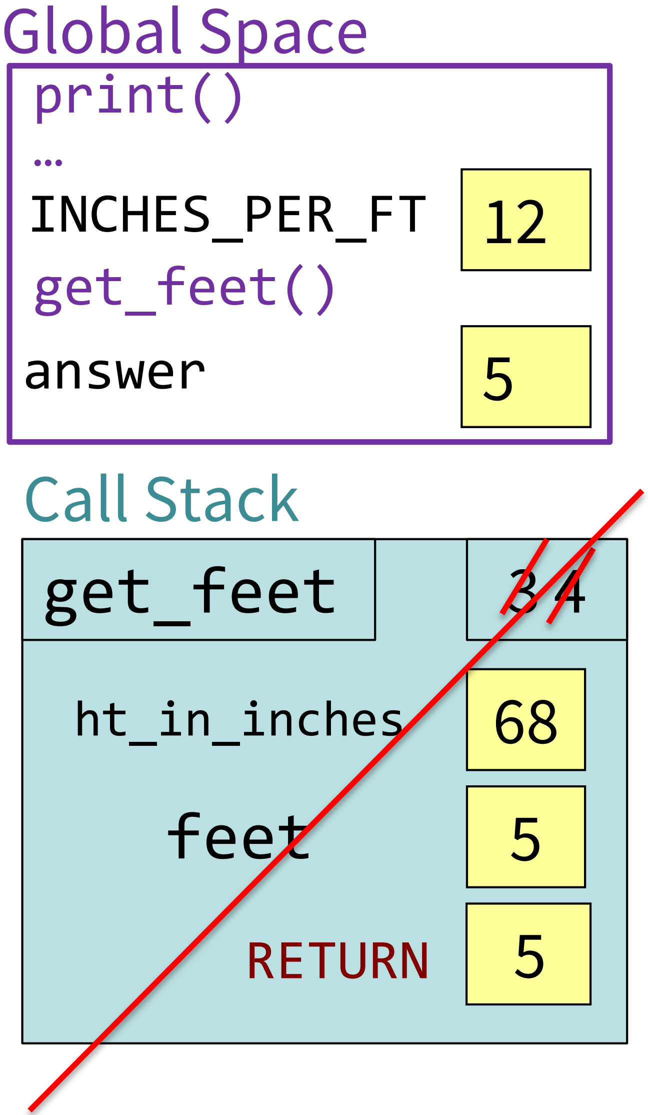
Python has just executed line 5.
A new global variable answer has been created.
The call frame for get_feet has been deleted.

Function Access to Global Space (8)

```
  # height3.py

1 INCHES_PER_FT = 12
2 def get_feet(ht_in_inches):
3     feet = ht_in_inches // INCHES_PER_FT
4     return feet

5 answer = get_feet(68)
6 print(answer)
```

Python has just executed line 6.

```
C:\> python height3.py
5
```



Global Space

print()	
…	
INCHES_PER_FT	12
get_feet()	
answer	5

Call Stack

get_feet	~~3~~ ~~4~~
ht_in_inches	68
feet	5
RETURN	5

Function Access to Global Space (9)

```python
# height3.py

INCHES_PER_FT = 12
def get_feet(ht_in_inches):
    feet = ht_in_inches // INCHES_PER_FT
    return feet

answer = get_feet(68)
print(answer)
```

```
C:\> python height3.py
5
C:\>
```

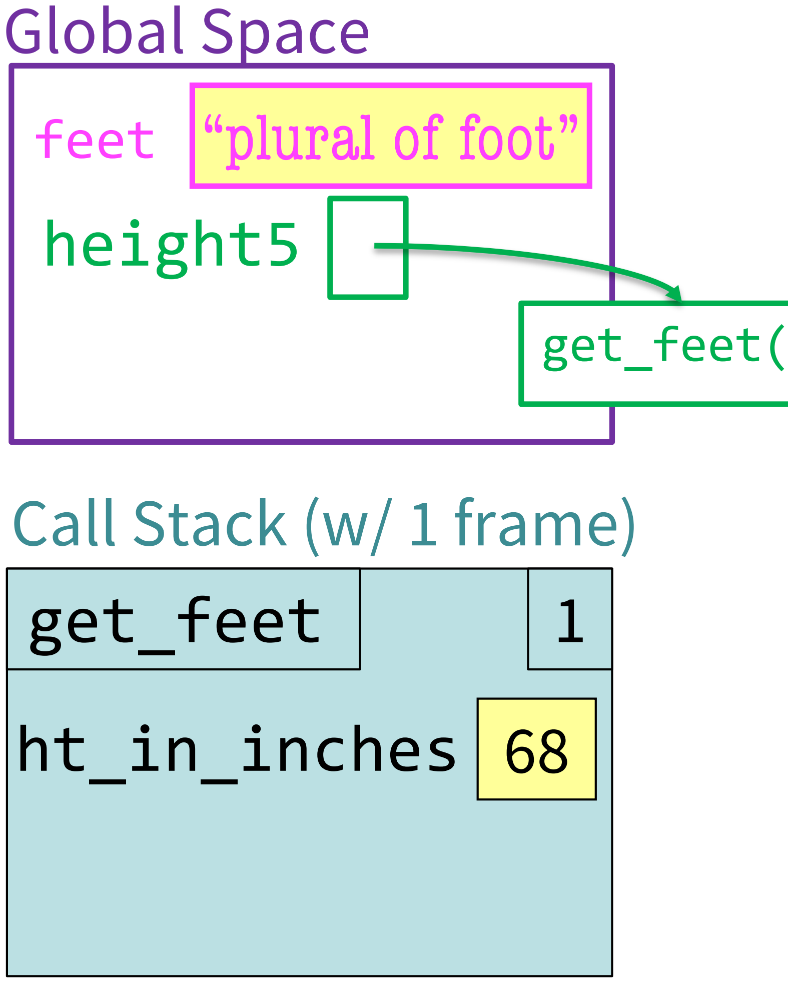
Python has completed executing all lines of the module. Python is no longer running, so the global space is gone. You can type a new command at the command line now.

Q: what about this??

What if a local variable inside a function has the same name as a global variable?

```
# height5.py


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

```
C:\> python
>>> feet = “plural of foot”
>>> import height5
>>> height5.get_feet(68)
```

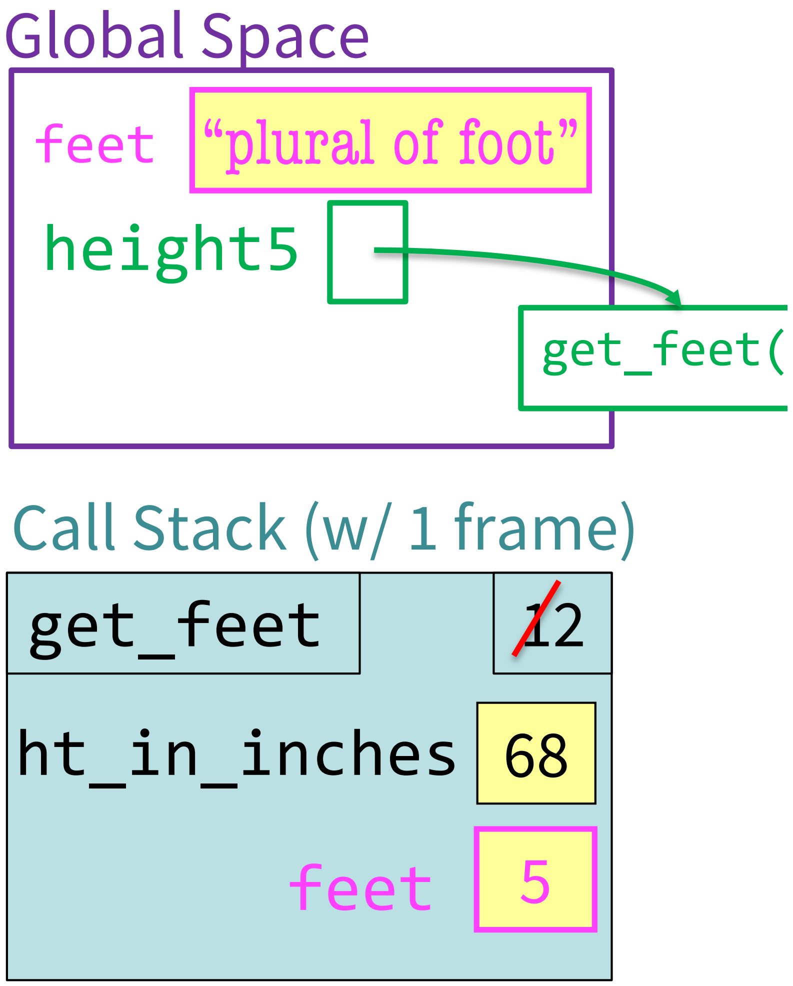


A: **Look, but don't touch!**

Can't change global variables in a function! Assignment to a global makes a new <u>local</u> variable!

```
# height5.py


def get_feet(ht_in_inches):

1    feet = ht_in_inches // 12

2    return feet
```

```
C:\> python
>>> feet = "plural of foot"
>>> import height5
>>> height5.get_feet(68)
```



Global Space

feet "plural of foot"

height5

get_feet(

Call Stack (w/ 1 frame)

get_feet ~~1~~2

ht_in_inches 68

feet 5

Use Python Tutor to help visualize

Lots of code for today:

https://www.cs.cornell.edu/courses/cs1110/2022sp/schedule/lecture/lec04/lec04.html

Paste it into the Python Tutor (http://cs1110.cs.cornell.edu/tutor/#mode=edit)

- Visualize the code as is
- Change the code
  - Try something new!
  - Insert an error! (misspell **`ht_in_inches`** or **`feet`**)
- Visualize again and see what is different

Call Frames and Global Variables

```python
# bad_swap.py
def swap(a,b):
   """Bad attempt at swapping
  globals a & b"""
   tmp = a
   a = b
   b = tmp

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

Question: Does this work?

What exactly gets swapped with function **swap**?

Paste this into the Python Tutor and see for yourself!

More Exercises (1)

Module Text

```
# my_module.py

def foo(x):
    return x+1


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

Python Interactive Mode

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

What does Python give me?

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

More Exercises (2)

Function Definition

```
# silly.py

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

Function Call

```
>>> import silly
>>> x = 2
>>> foo(3,4)
>>> x
…
```

What does Python give me?

A: 2
B: 3
C: 16
D: Nothing
E: I do not know

More Exercises (3)

Module Text

```
# module.py


def foo(x):
     x = 1+2
     x = 3*x
```

Python Interactive Mode

```
>>> import module
>>> module.x
…
```

What does Python give me?

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

More Exercises (4)

Module Text

```
# module.py

def foo(x):
      x = 1+2
      x = 3*x

x = foo(0)
```

Python Interactive Mode

```
>>> import module
>>> module.x
…
```

What does Python give me?

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

More Exercises (5)

Module Text

```
# module.py

def foo(x):
     x = 1+2
     x = 3*x
     return x+1

x = foo(0)
```

Python Interactive Mode

```
>>> import module
>>> module.x
…
```

What does Python give me?

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