http://www.cs.cornell.edu/courses/cs1110/2021sp

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]

Review ideas from previous lecture

Module vs. Script

`print` statement

Running a Script

- From the command line, type:

  python <script filename>

- Example:

  C:\> python my_module.py

  C:\>

looks like nothing happened

- Actually, something did happen
  - Python executed all of my_module.py

Running my_module.py as a script

my_module.py

Command Line

```
# my_module.py
```

Python does not execute (because of #)

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

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

Python does not execute (because of """ and """)

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

Python executes this.

Python executes this.

x 9

Clicker Question

fah2cel.py

```
# fah2cel.py


"""Convert 32 degrees
Fahrenheit
to degrees Celsius"""

f= 32.0
c= (f-32)*5/9
```

Command Line

```
C:\> python fah2cel.py
C:\> fah2cel.c
```

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

(A) >>>

(B) 0.0
>>>

(C) an error message

(D) The text of fah2cel.py

(E) Sorry, no clue.

Running fah2cel.py as a script

fah2cel.py

```
# fah2cel.py

"""Convert 32 degrees Fahrenheit
to degrees Celsius"""

f= 32.0
c= (f-32)*5/9
```

Command Line

```
C:\> python fah2cel.py
C:\>
```

when the script ends, all memory used by fah2cel.py is deleted

thus, all variables get deleted (including f and c)

so there is no evidence that the script ran

Modules vs. Scripts

Module

- Provides functions, variables
- import it into Python shell

⇨ Within Python shell you have access to the functions and variables of the imported module

Script

- Behaves like an application
- Run it from command line

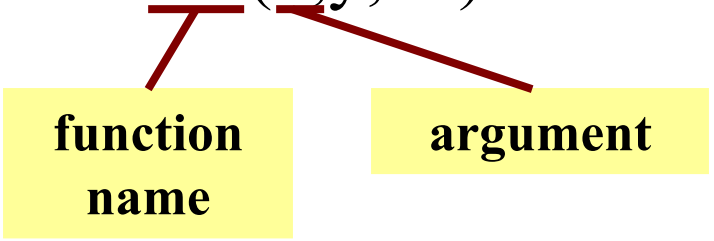
⇨ After running the app you're back at the command line (not in Python shell)

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

Defining our own functions

From last time: Function Calls

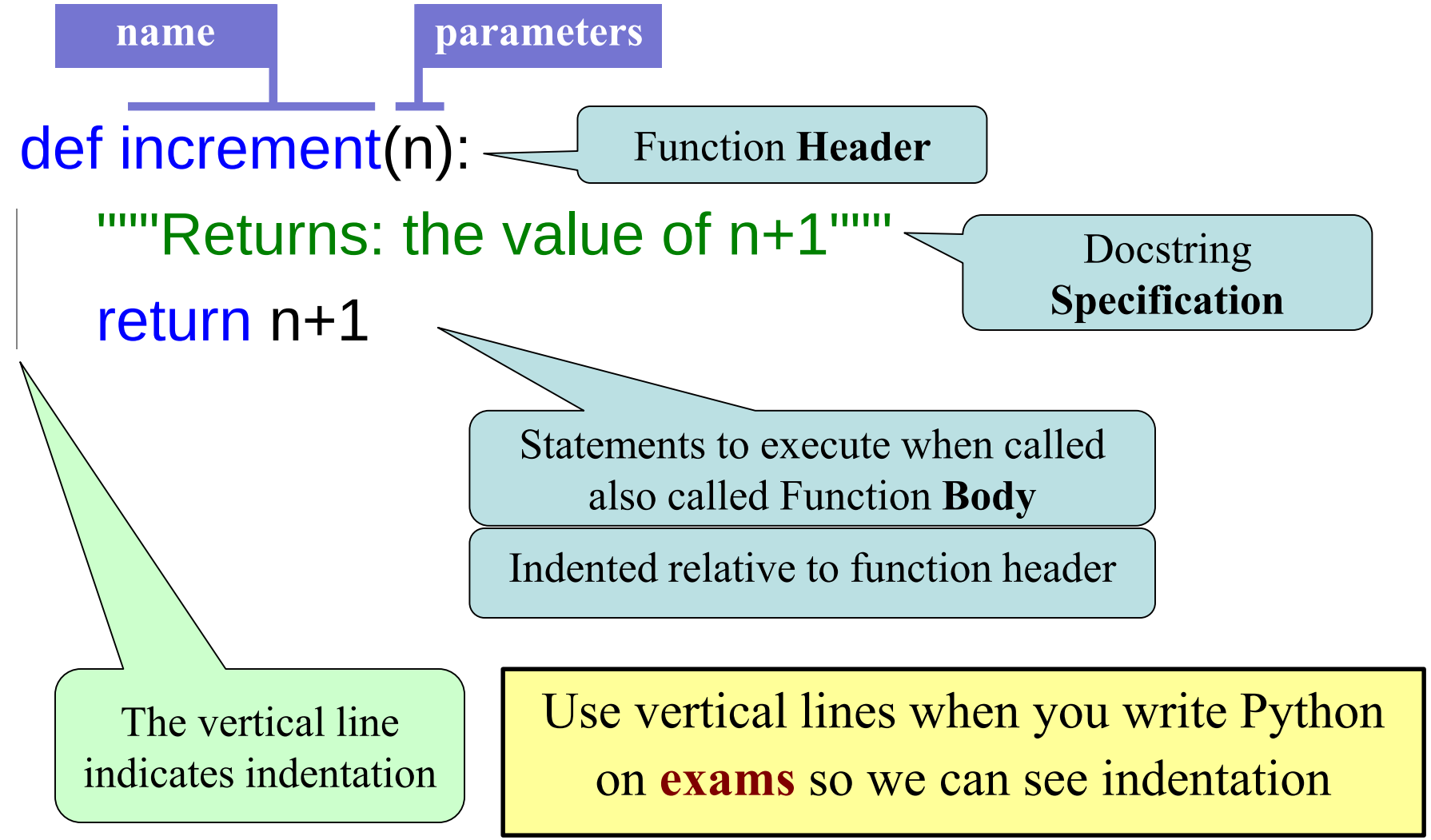
- Function expressions have the form **fun**(x,y,…)



- **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



The `return` Statement

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

Organization of a Module

```
# simple_math.py


def increment(n):
    return n+1



increment(2)
```

- 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

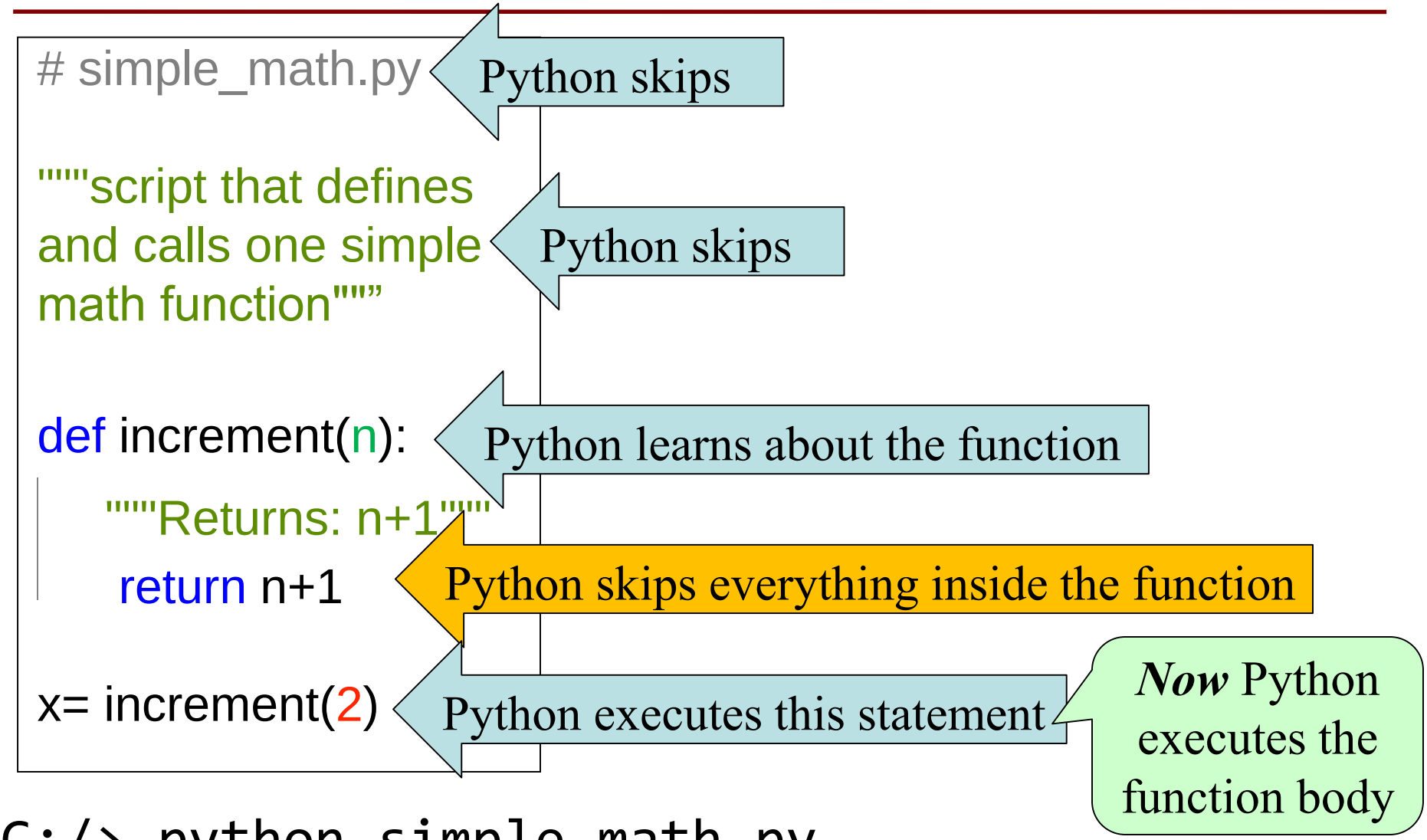
Function definition

- Defines what the function **will do**
- Declaration of parameters, **n** in this case
- **Parameter:** the variable that is listed within the parentheses of a function header.

Function call

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

Executing the script simple_math.py



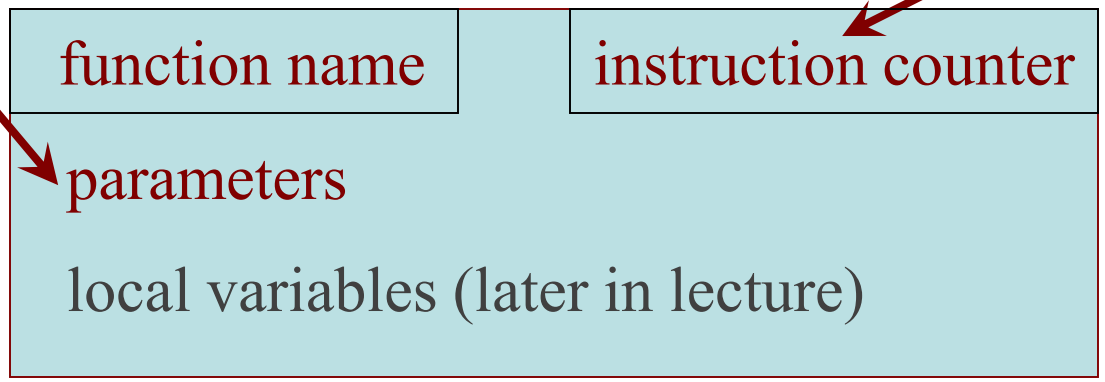

```
C:/> python simple_math.py
```

Understanding How Functions Work

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

Draw parameters as variables (named boxes)

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



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)
```

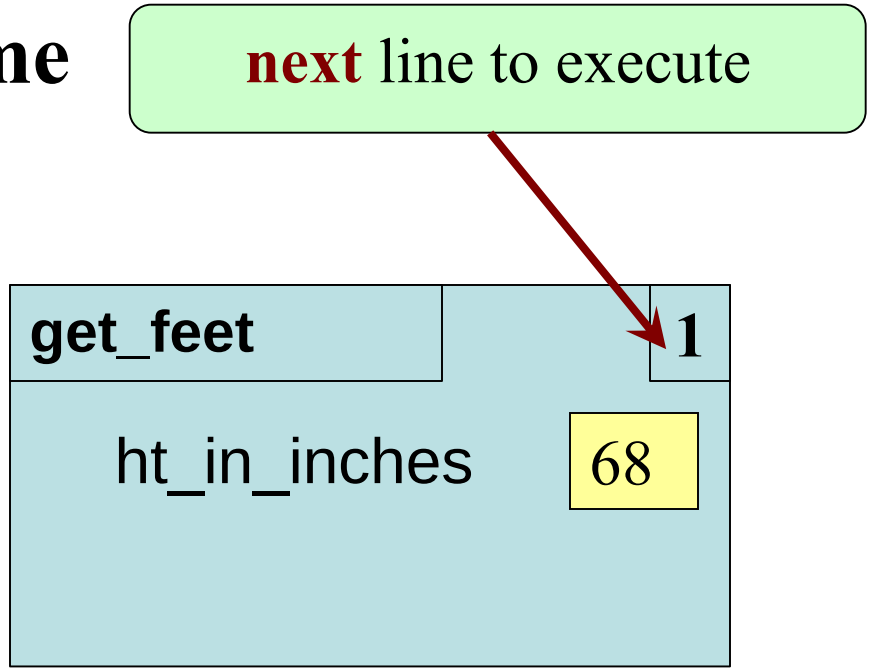
```
def get_feet(ht_in_inches):
    return ht_in_inches // 12
```

1

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



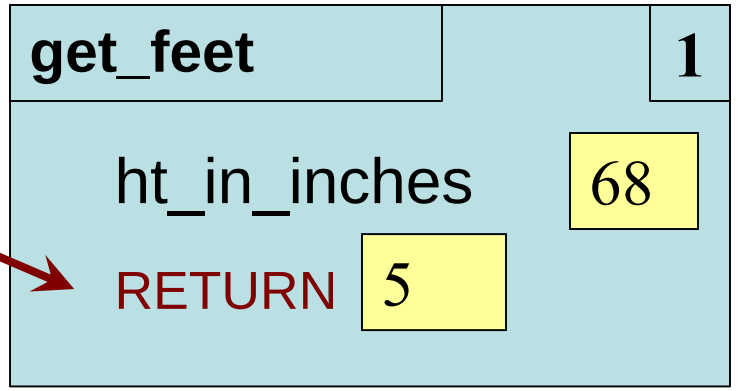

```
  def get_feet(ht_in_inches):
1     return ht_in_inches // 12
```

Example: get_feet(68)

PHASE 2:

Execute function body

Return statement creates a special variable for result



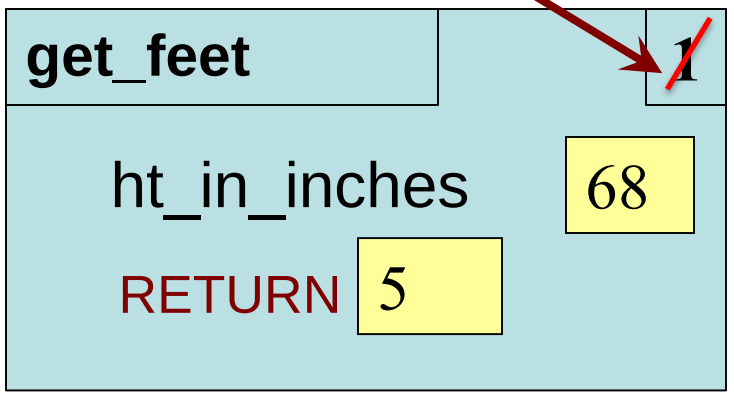

```
def get_feet(ht_in_inches):
1   return ht_in_inches // 12
```

Example: get_feet(68)

PHASE 2:

Execute function body

The return terminates;
no next line to execute

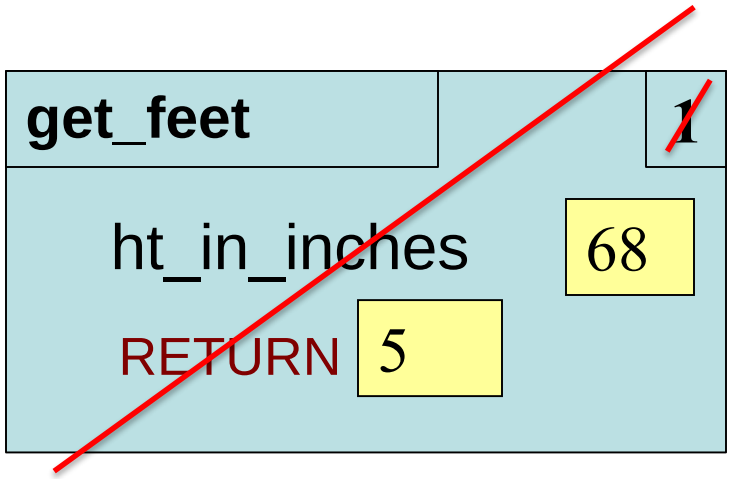



```
def get_feet(ht_in_inches):
    return ht_in_inches // 12
```

1

Example: get_feet(68)

PHASE 3: Delete (cross out) call frame



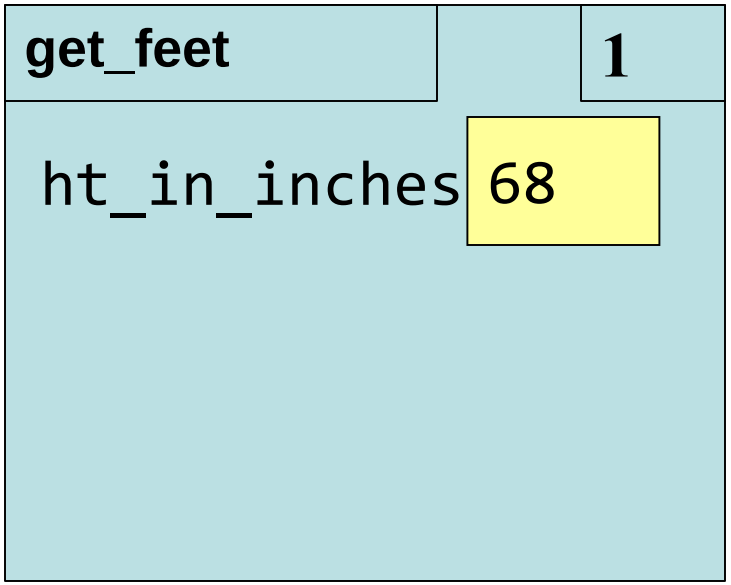

```
def get_feet(ht_in_inches):
1       return ht_in_inches // 12
```

Local Variables (1)

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

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

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

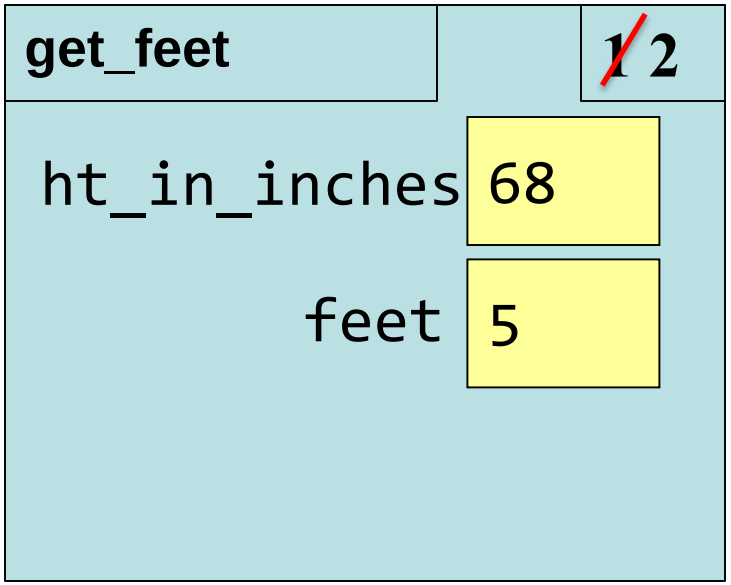


Local Variables (2)

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

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

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

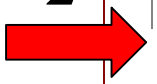


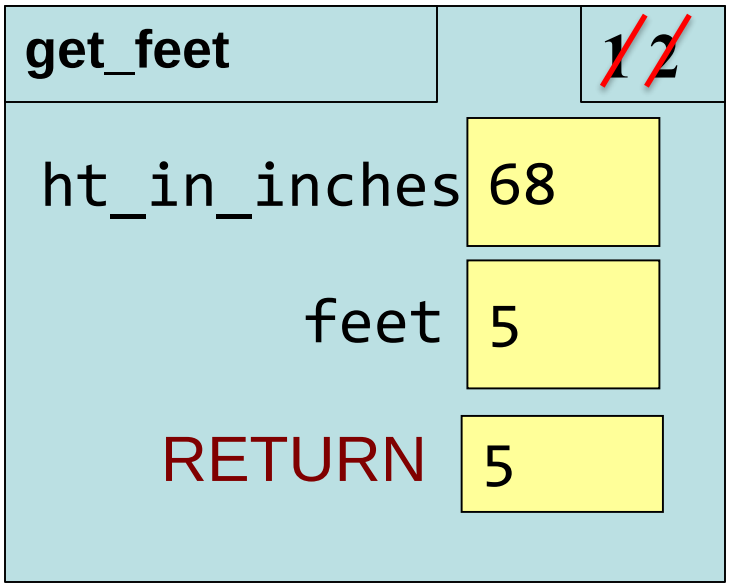
Local Variables (3)

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

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

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





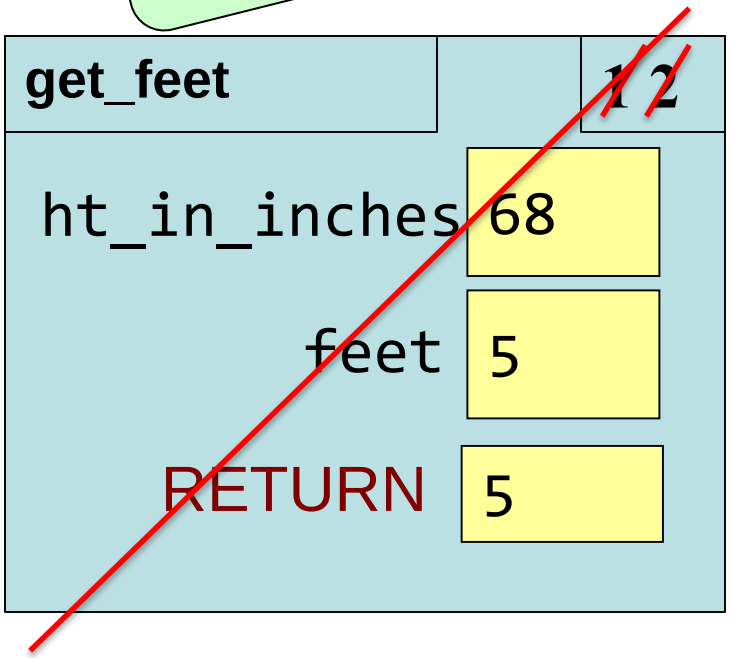
Local Variables (4)

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

```
>>> import height2
>>> height2.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
```

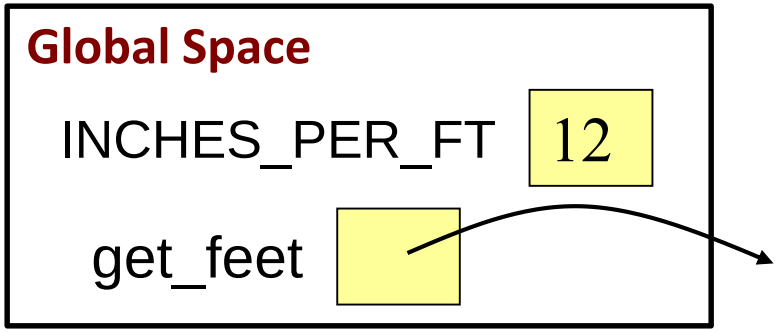
Function Call

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

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

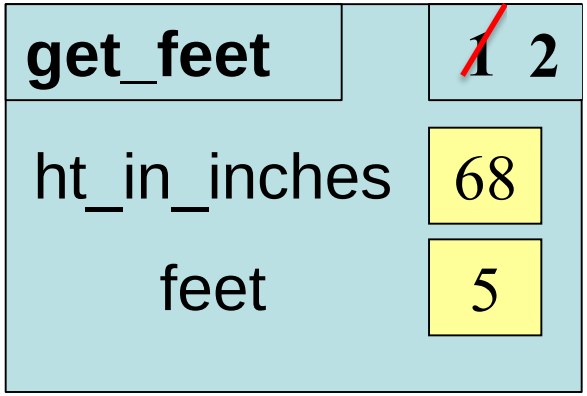
Function Access to Global Space

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



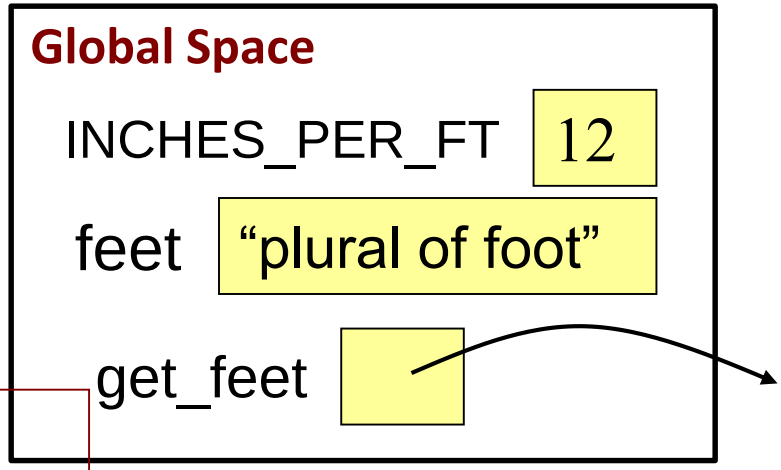

```
INCHES_PER_FT = 12


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

get_feet(68)
```

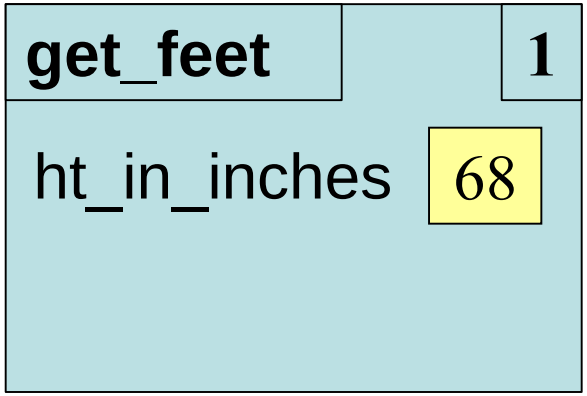
1
2



What about this??

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



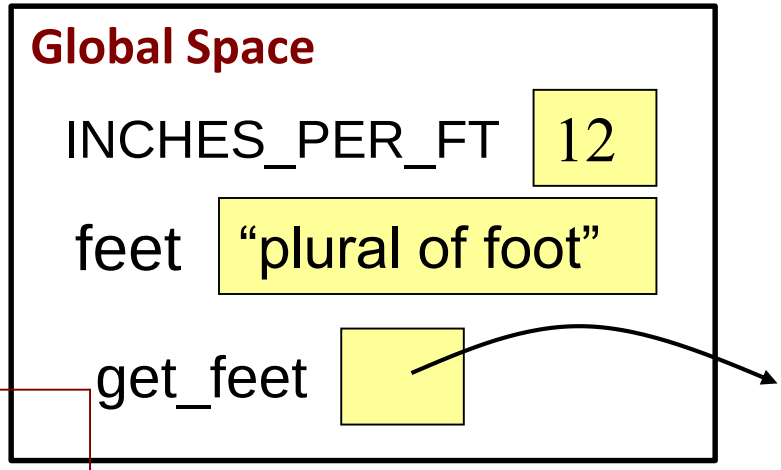



```
INCHES_PER_FT = 12
feet = "plural of foot"

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!

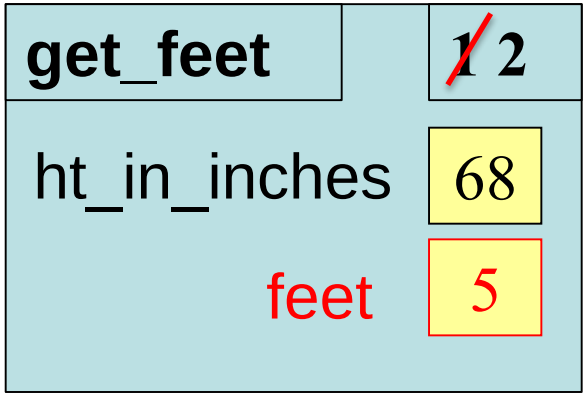
Can't change global variables

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):
    feet = ht_in_inches // INCHES_PER_FT
    return feet

get_feet(68)
```

1
2



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)
```

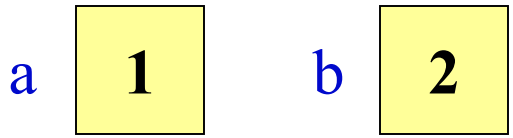
1. Visualize code as is
2. Change code to introduce an error, e.g. misspell **ht_in_inches**. Visualize again.

Call Frames and Global Variables

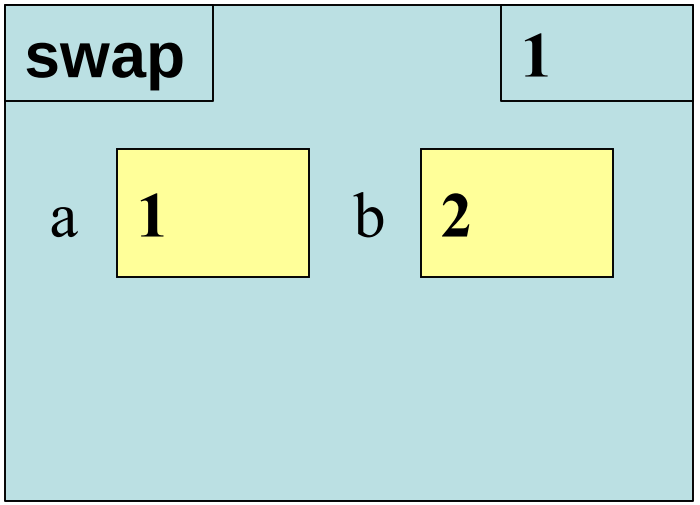
```
def swap(a,b):
    """Swap global a & b"""
1   tmp = a
2   a = b
3   b = tmp
```

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

Global Variables



Call Frame



Question: What exactly gets swapped with function swap?

More Exercises (1)

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

More Exercises (2)

Function Definition

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

Function Call

```
>>> 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
```

Interactive Python

```
>>> 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)
```

Interactive Python

```
>>> 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)
```

Interactive Python

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

What does Python give me?

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