Module 12

Python Memory

The Problem of Methods

- Introduced objects in previous video seires
  - “Folders” with variables and functions
  - Called **attributes** and **methods**
- But we saw that strings also have methods

string.name(x,y,…)

argument

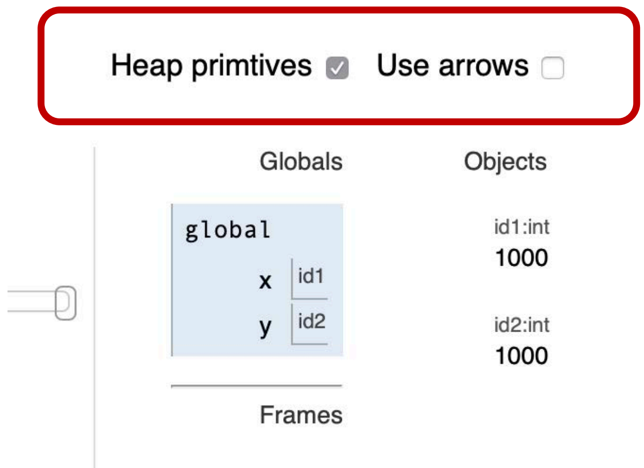
method name

arguments

Are strings objects?

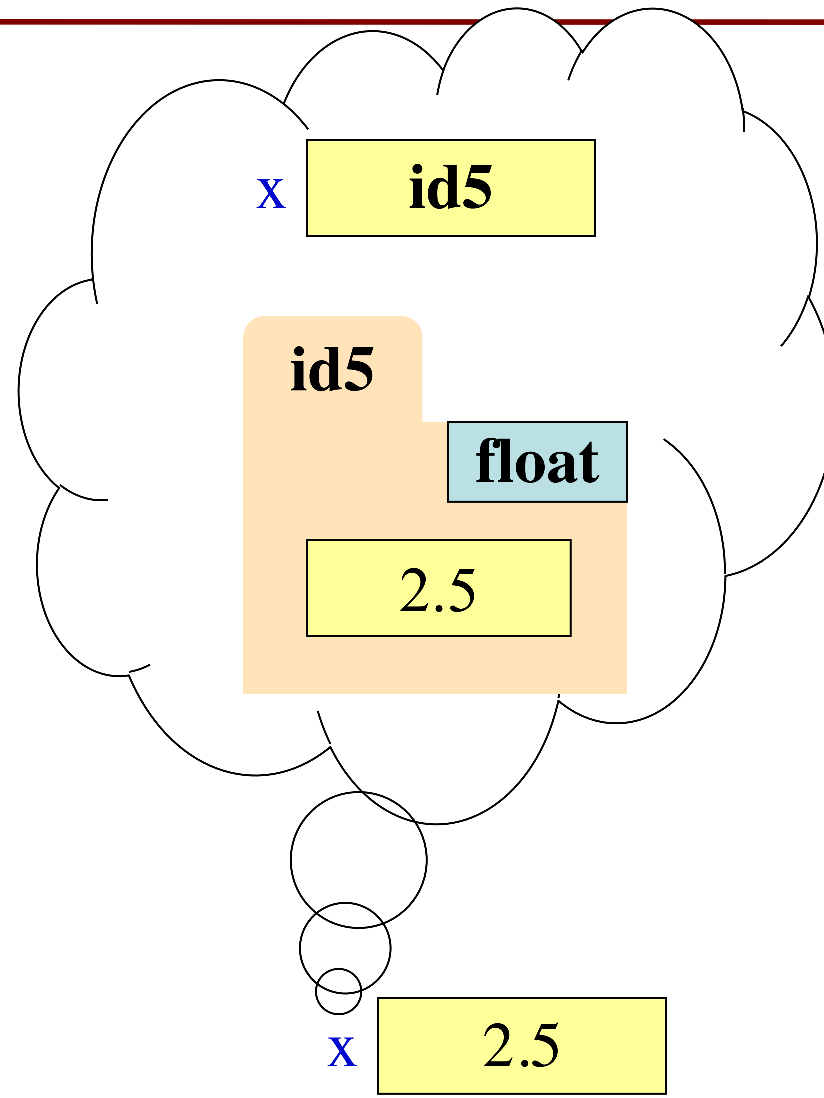
Surprise: All Values are in Objects!

- Including basic values
  - int, float, bool, str



- **Example**:

```
>>> x = 1000
>>> id(x)
```



This Explains A Lot of Things

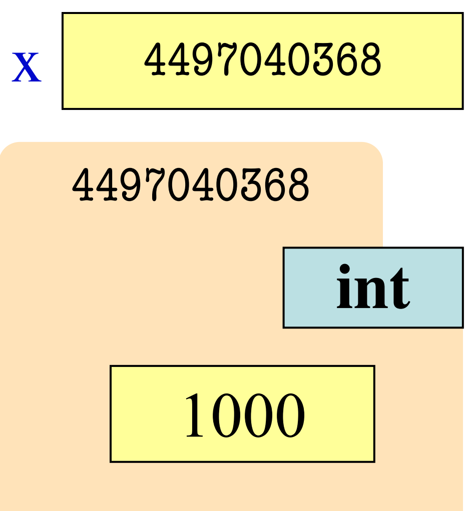
- Primitives act like classes
  - Conversion function is really a constructor
  - Remember constructor, type have same name
- Example:

```
>>> type(1)
<class 'int'>
>>> int('1')
1
```

- Design goals of Python 3
  - Wanted everything an object
  - Makes processing cleaner
- But makes learning harder
  - Objects are complex topic
  - Want to delay if possible

But Not Helpful to Think This Way

- Number folders are **immutable**
  - "Attributes" have no names
  - No way to reach in folder
  - No way to change contents



Makes a brand new int folder

```
>>> x = 1000
>>> y = 1000
>>> id(x)
4497040368
>>> id(y)
4497040400
>>> y = y+1
>>> id(y)
4497040432
```

But Not Helpful to Think This Way

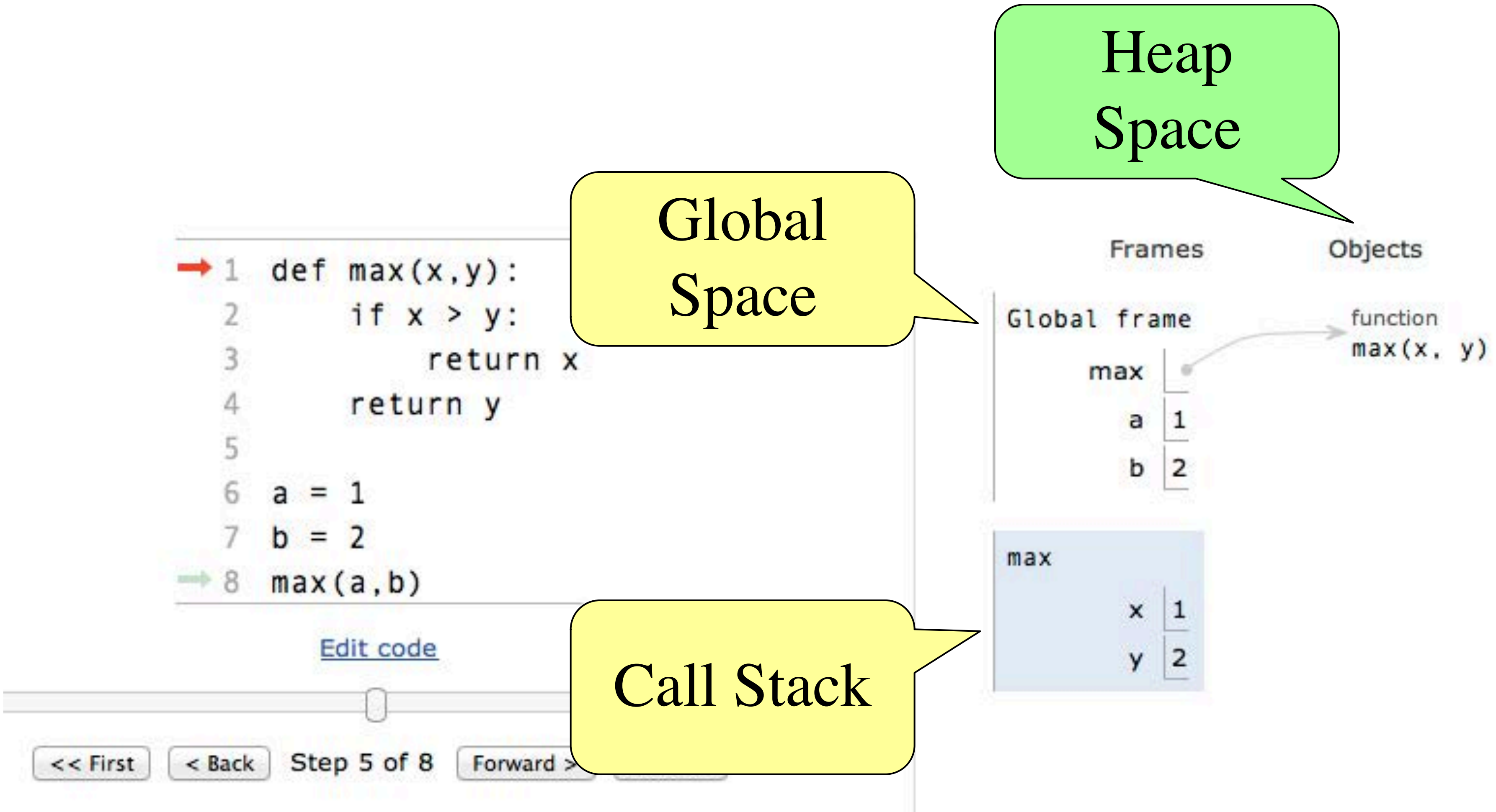
- Number folders are **immutable**
  - "Attributes" have no names
  - No way to reach in folder
  - No way to change contents
- Remember **purpose of folder**
  - Show how objects can be altered
  - Show how variables "share" data
  - This **cannot happen** in basic types
- So just **ignore the folders**
  - (The are just metaphors anyway)

```
>>> x = 1000
>>> y = 1000
>>> id(x)
4497040368
>>> id(y)
4497040400
>>> y = y+1
>>> id(y)
4497040432
```

Why Show All This?

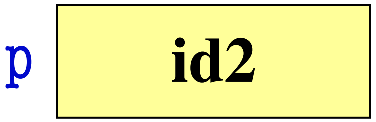
- Many of these are **advanced topics**
  - Only advanced programmers need
  - Will never need in the context of 1110
- But you might use them by *accident*
- **Goal: Teach you to read error messages**
  - Need to understand what messages say
  - Only way to debug your own code

The Three “Areas” of Memory



Global Space

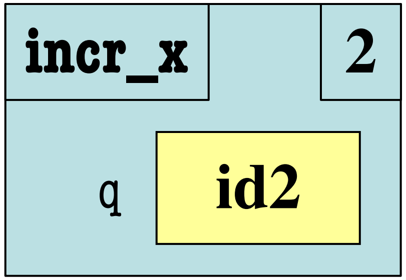
- This is the **area you "start with"**
  - First memory area you learned to visualize
  - A place to store "global variables"
  - Lasts until you quit Python



- What are **global variables**?
  - **Any assignment not in a function definition**
  - Also **modules & functions!**
  - Will see more on this in a bit

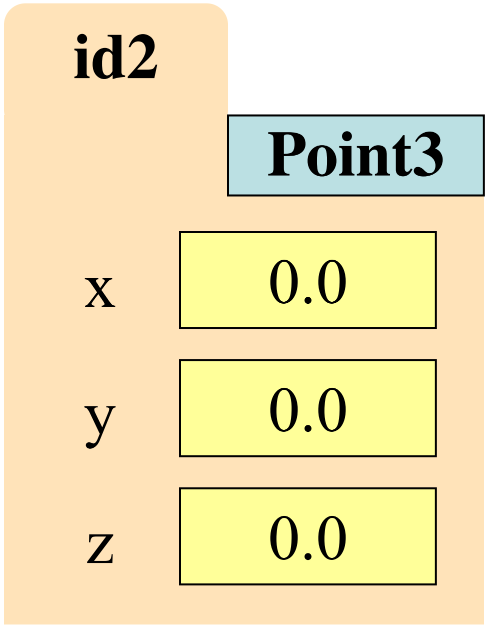
The Call Stack

- The area **where call frames live**
  - Call frames are created on a function call
  - May be several frames (functions call functions)
  - Each frame deleted as the call completes
- Area of volatile, temporary memory
  - Less permanent than global space
  - Think of as “scratch” space
- Primary focus of Assignment 2



Heap Space or "The Heap"

- **Where the "folders" live**
  - Stores *only* folders
- Can only **access indirectly**
  - Must have a variable with identifier
  - Can be in global space, call stack
- MUST have **variable with id**
  - If no variable has id, it is *forgotten*
  - Disappears in Tutor immediately
  - But not necessarily in practice
  - Role of the *garbage collector*



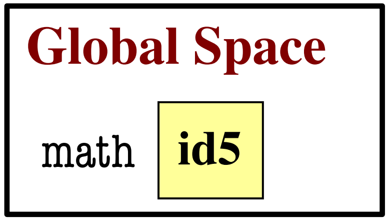
Revisiting Modules

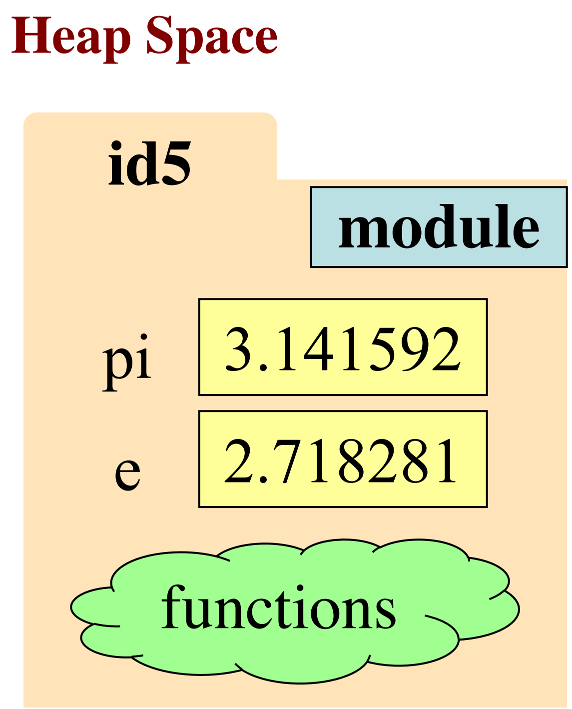
- Modules seem to behave a lot like objects
  - They can have *variables*: `math.pi`
  - Can even reassign these variables!
  - Function calls look like *methods*: `math.cos(1)`
- So are they also objects?
  - Said everything in Python is an object
- **Yes** (sort of)
  - Look same in memory, but created differently
  - Need to understand what happens on import

Modules and Global Space

- Importing a module:
  - Creates a global variable (same name as module)
  - Puts contents in a **folder**
    - Module variables
    - Module functions
  - Puts folder id in variable
- Can reassign module var
- Tutor won't show contents

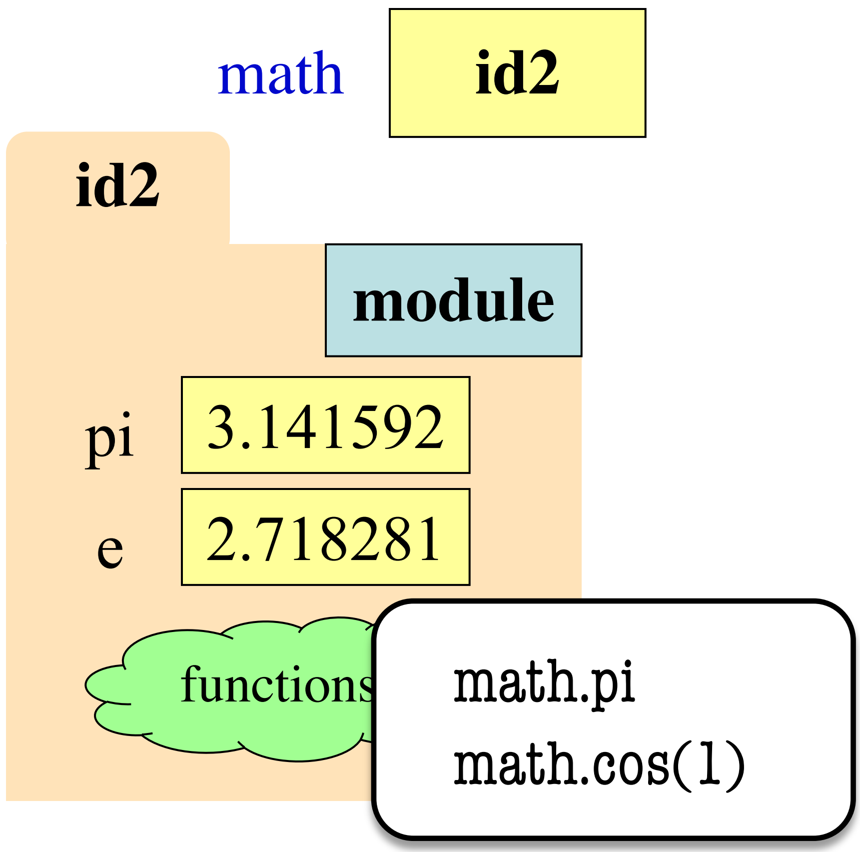
```
import math
```



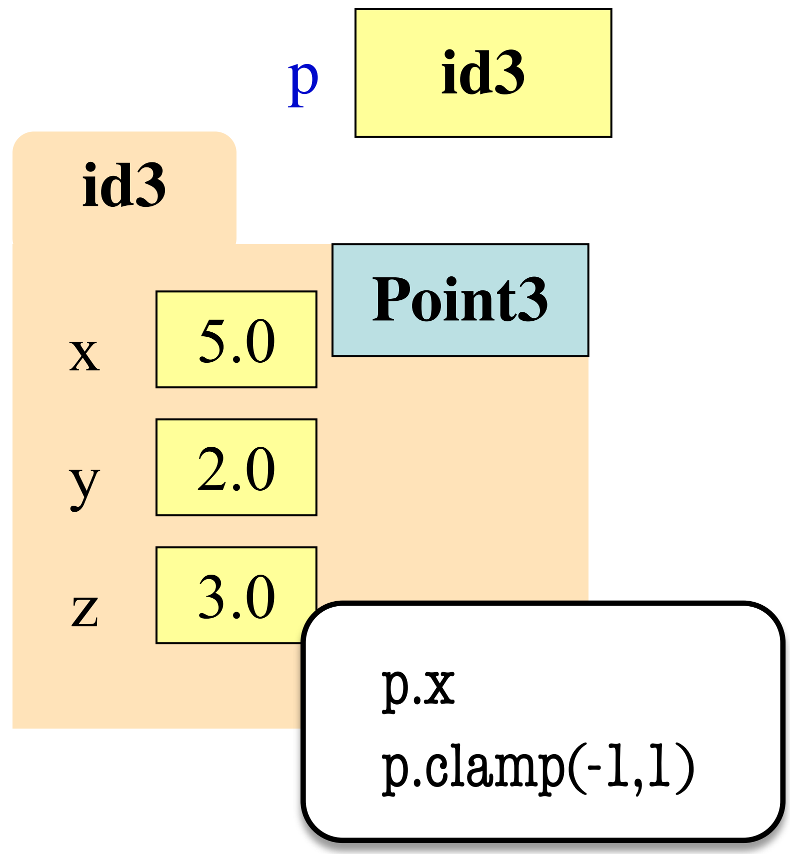


Modules vs Objects

Module



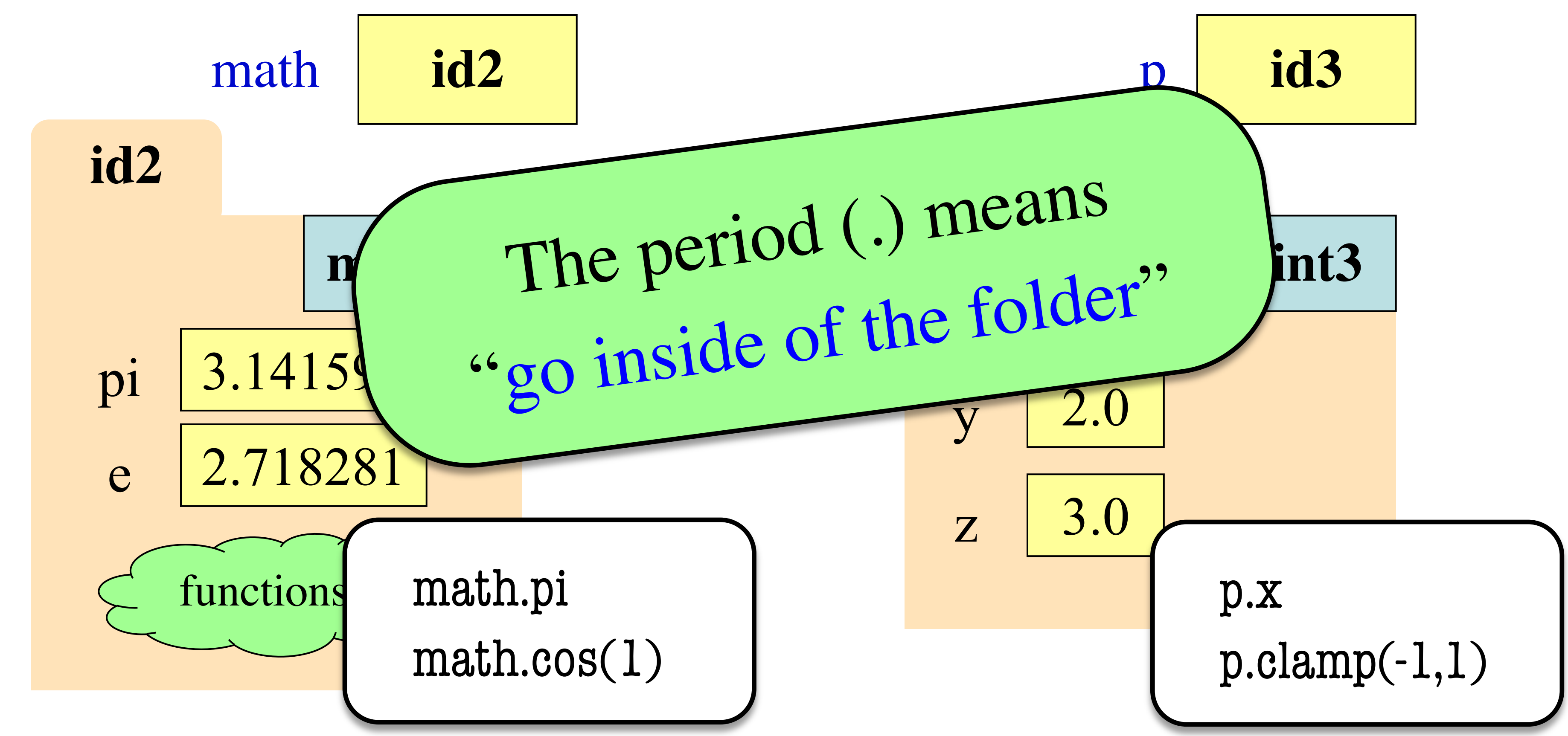
Object



Modules vs Objects

Module

Object



So Why Have Both?

- Question is a matter of program design
  - Some software will use modules like objects
- Classes can have **many instances**
  - Infinitely many objects for the `Point3` class
  - Reason we need a constructor function
- Each module is **a unique instance**
  - Only one possibility for `pi`, `cosine`
  - That is why we import them
  - Sometimes refer to as *singleton* objects

So Why Have Both?

- Question is a matter of program design
  - Some software will use modules like objects
- Classes can have **many instanc**
  - Infinitely m
  - Re
- Each **nstance**
  - Only possibility for `pi`, `cosine`
  - That is why we import them
  - Sometimes refer to as *singleton* objects

Choice is an advanced topic
beyond scope of this course

Are Functions Objects?

- "Everything an object" has major ramifications
  - Forced us to completely rethink modules
  - Anything else? What about functions?
- But functions live in the call stack!
  - Function **calls** live in the call stack
  - Remember there are two parts to a function
  - Where does the function *definition* live?
  - Python had to store the code somewhere
- If you are thinking objects, you are right

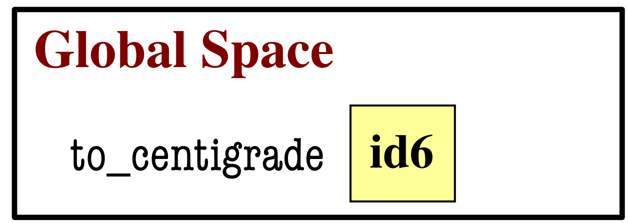
Functions and Global Space

- A function definition…
  - Creates a global variable (same name as function)
  - Creates a **folder** for body
  - Puts folder id in variable
- Variable vs. Call

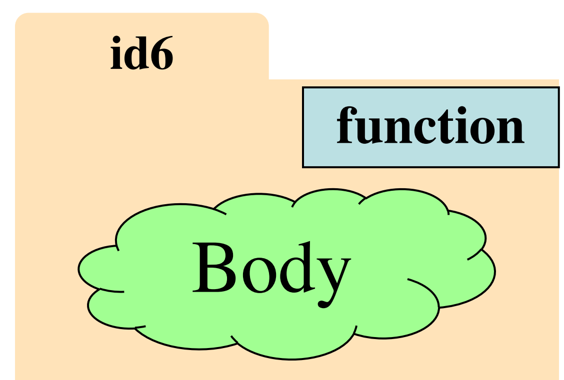
```
>>> to_centigrade
<fun to_centigrade at 0x100498de8>
>>> to_centigrade (32)
0.0
```

```
def to_centigrade(x):
    return 5*(x-32)/9.0
```

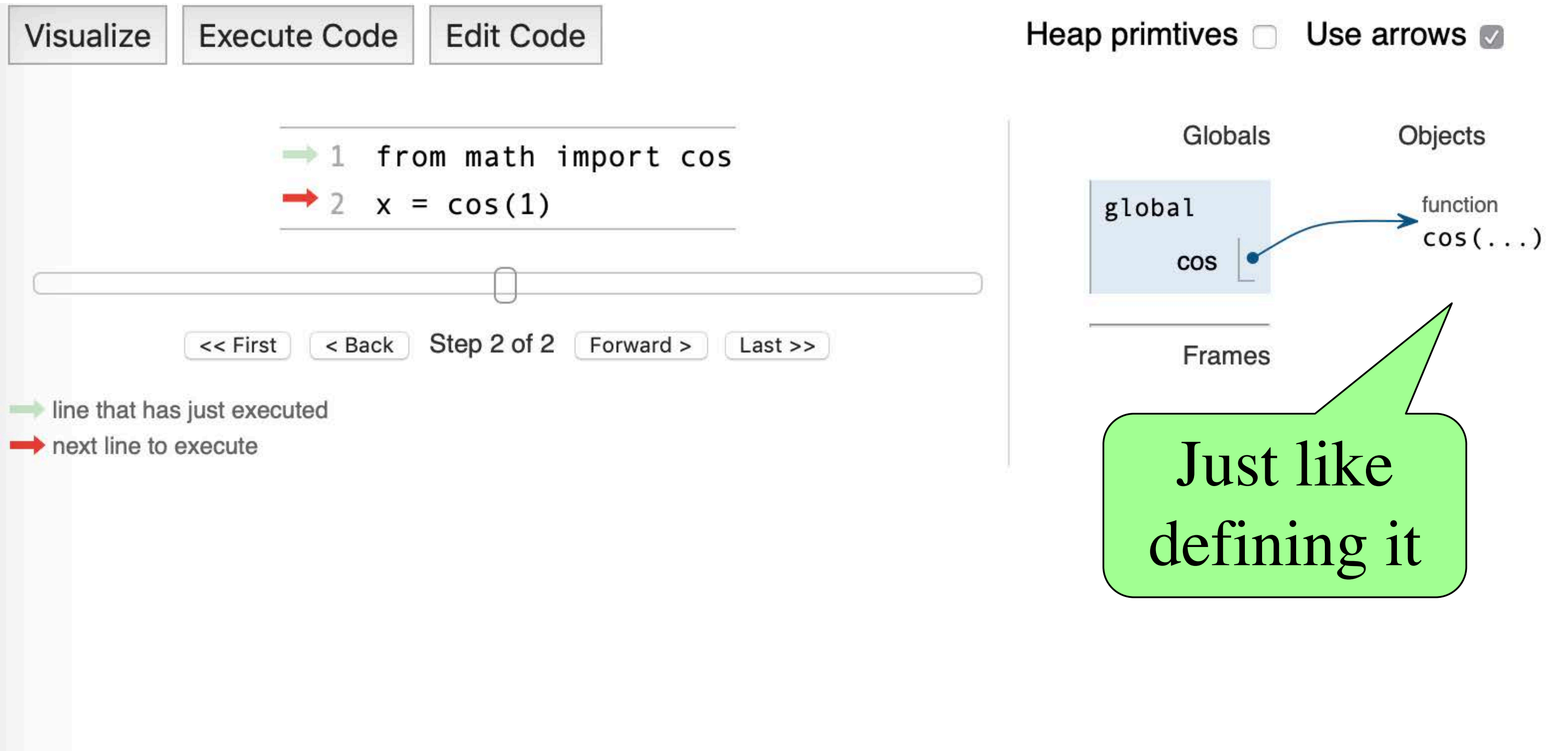
Body



Heap Space



What Does Importing a Function Do?



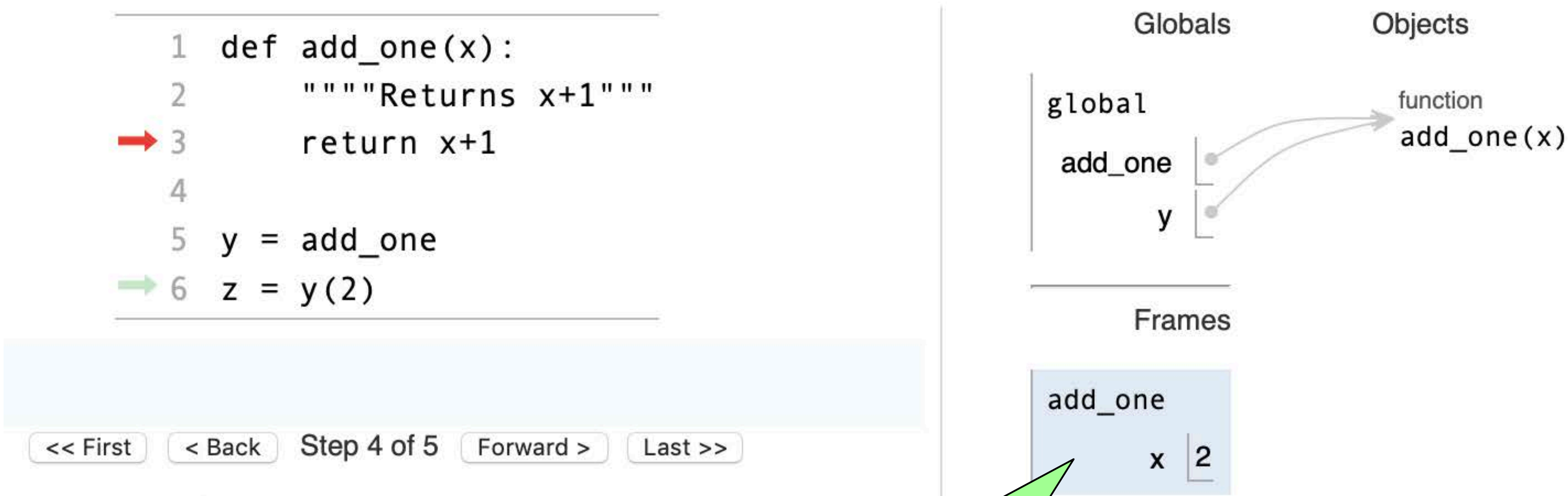
How About **import** *?

Ouch!

Working with Function Variables

- So function definitions are objects
  - Function names are just variables
  - Variable refers to a folder storing the code
  - If you reassign the variable, it is lost
- You can also assign them to other variables
  - Variable now refers to that function
  - You can use that **NEW** variable to call it
  - Just use variable in place of function name

Example: add_one



Frame remembers the original name

Application: Functions as Parameters

```
def doit(f,arg):
    """Returns the result of the call f(arg)

    Param: f the function to call
    Precond: f a function that takes one argument

    Param arg: the function argument
    Precond: arg satisfies the precondition of f"""
    return f(arg)
```

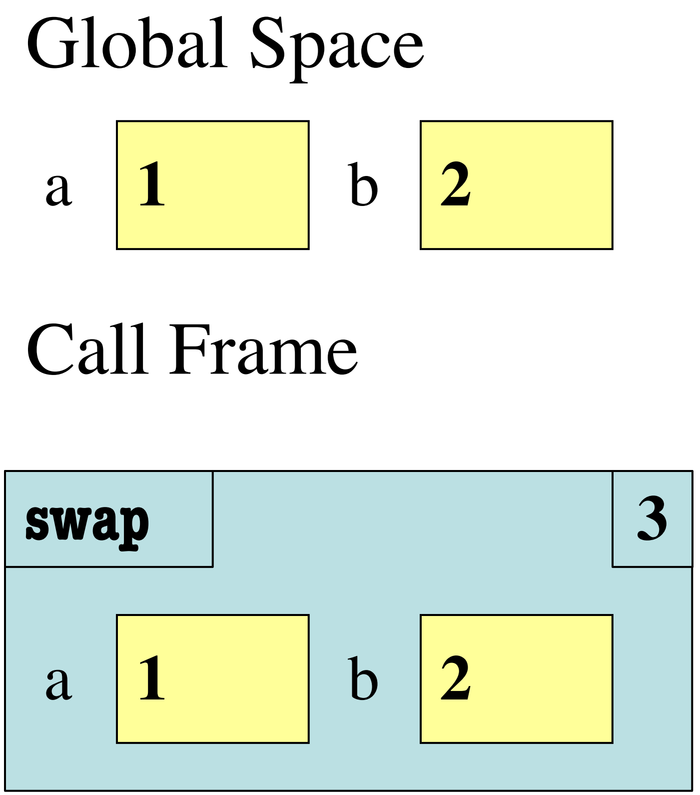
Will see practical applications of this in a later video series

Call Frames vs. Global Variables

The function does **nothing**:

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

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

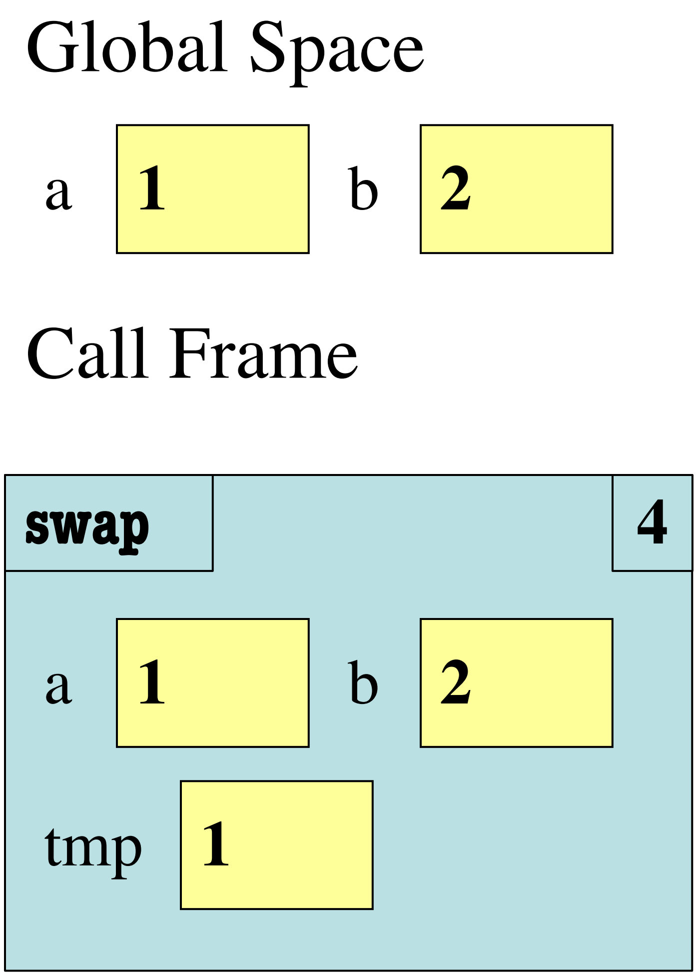


Call Frames vs. Global Variables

The function does **nothing**:

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

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

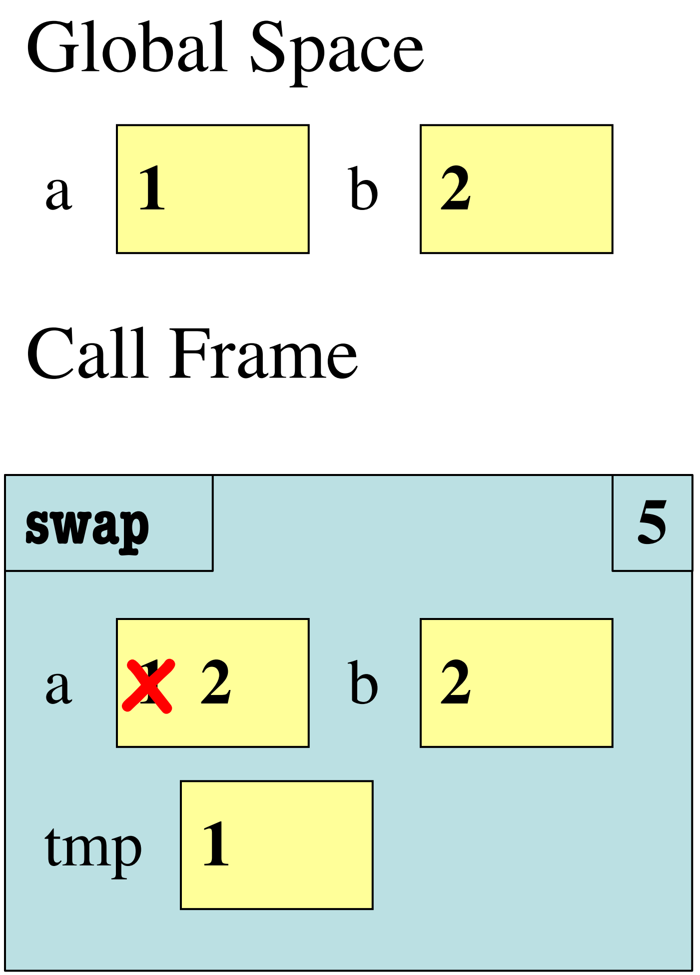


Call Frames vs. Global Variables

The function does **nothing**:

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

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

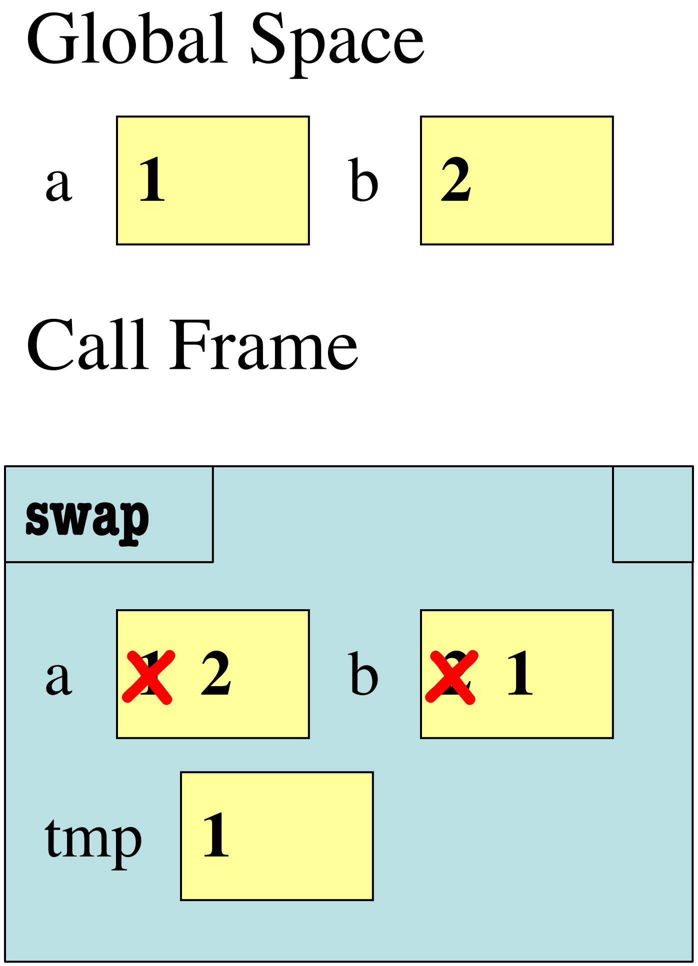


Call Frames vs. Global Variables

The function does **nothing**:

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

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



Call Frames vs. Global Variables

The function does **nothing**:

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

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

Global Space

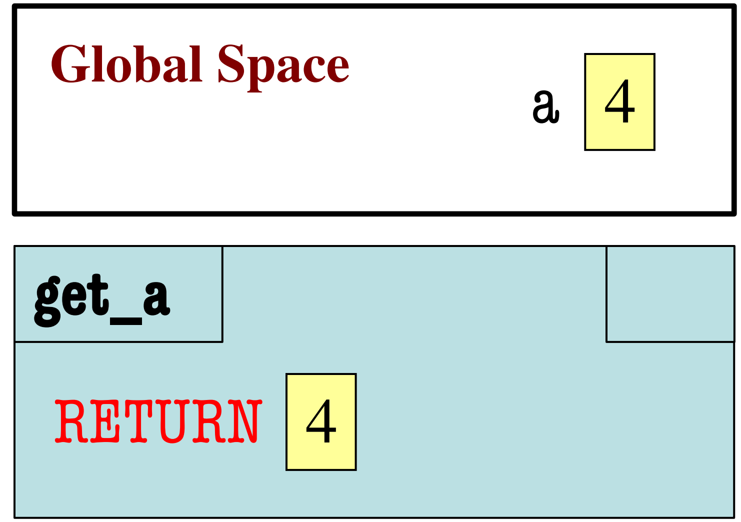
a **1** b **2**

Call Frame

ERASE THE FRAME

Functions Can Access Global Space

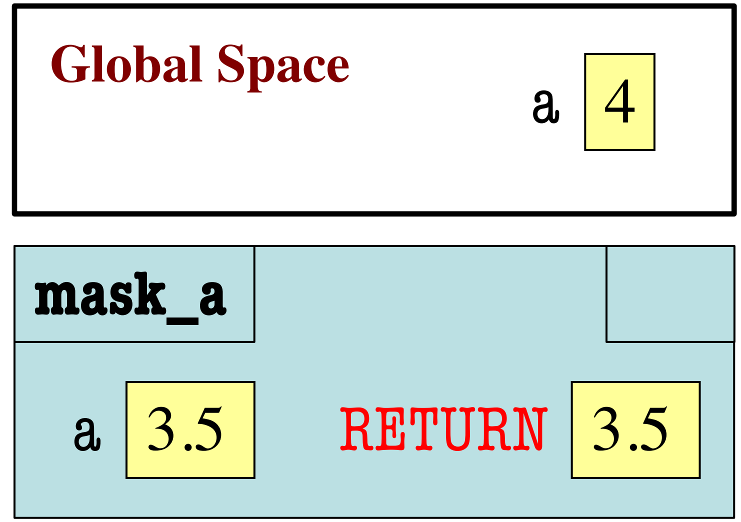
- Ways to use a global
  - Have to use in expression
  - **CANNOT** do assignment
- What happens if assign?
  - Makes a new local instead
  - Even if you assign it later
- So what use for globals?
  - Typically use as *constants*
  - **Example**: math.pi




```
a = 4 # global var
...
def get_a():
...    """..."""
    return a # global
```

Functions Can Access Global Space

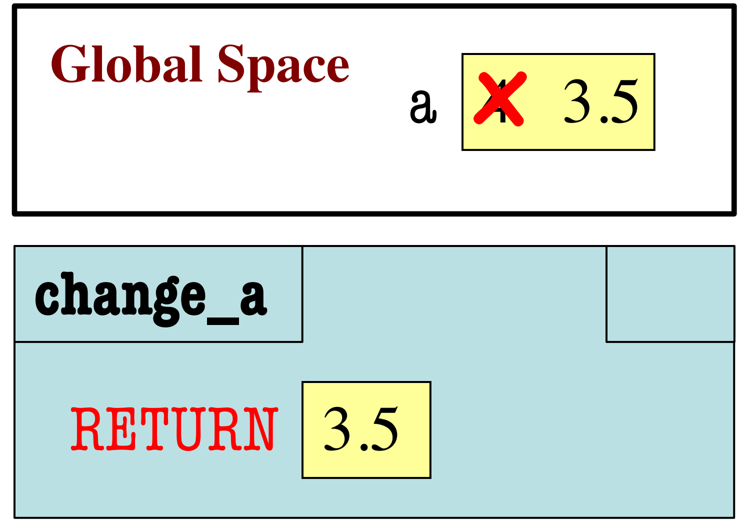
- Ways to use a global
  - Have to use in expression
  - **CANNOT** do assignment
- What happens if assign?
  - Makes a new local instead
  - Even if you assign it later
- So what use for globals?
  - Typically use as *constants*
  - **Example**: math.pi




```
def mask_a():
    """..."""
...
    a = 3.5
    return a # local
```

The Global Keyword

- Possible to change global
  - Have to mark it as such
  - global <variable>
  - Should be at body start
- Use sparingly
  - Using globals is confusing
  - Easy to get lost
  - Best for constants



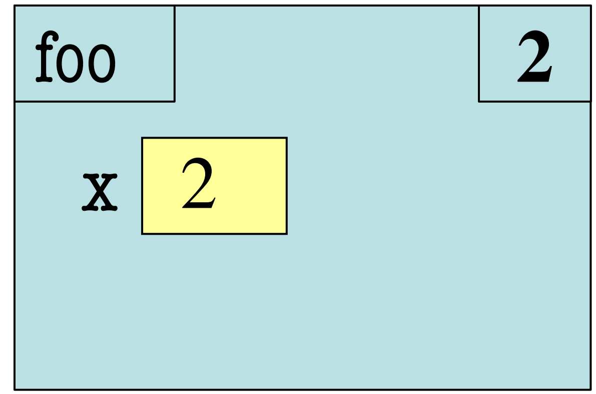

```
def change_a():
    """..."""
    global a
    a = 3.5
    return a # local
```

Function Bodies Can Contain Other Calls

- We have seen this with print in greet
  - Does print have a call frame?
  - Yes, but cannot visualize (definition hidden)
- What happens when one calls another?
  - Have to create a new call frame
  - Old call frame **freezes** in place
  - Waits until second frame is erased
  - Then first frame continues again

One Function Calling Another

```
def foo(x):
    y = x+1
    z = bar(y)
    return z

def bar(x):
    y = x-1
    return y

w = foo(2)
```

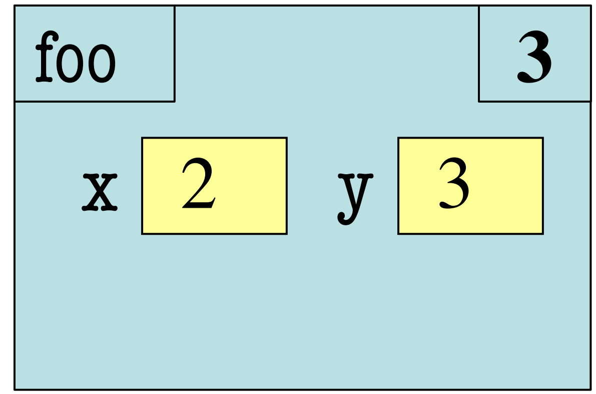


Let's visualize ourselves first.
(Tutor incomplete)

One Function Calling Another

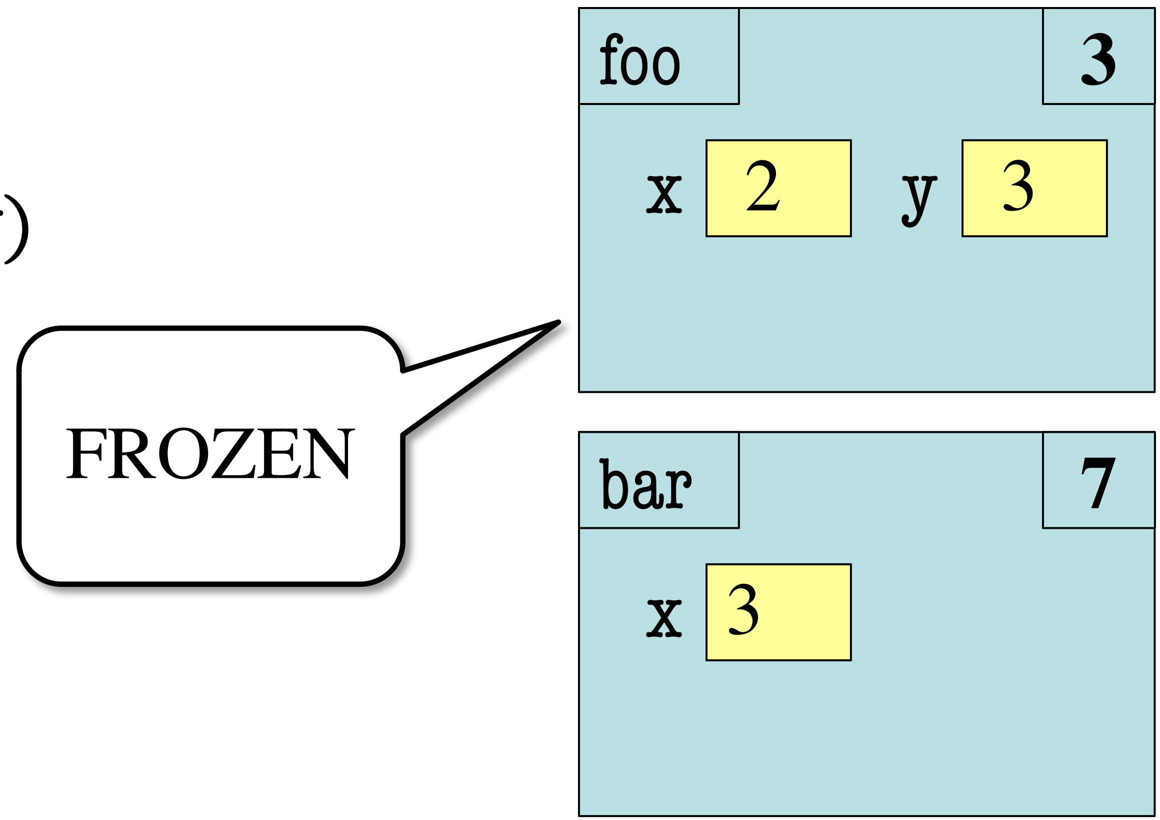
```
def foo(x):
    y = x+1
    z = bar(y)
    return z

def bar(x):
    y = x-1
    return y

w = foo(2)
```

Ready to execute



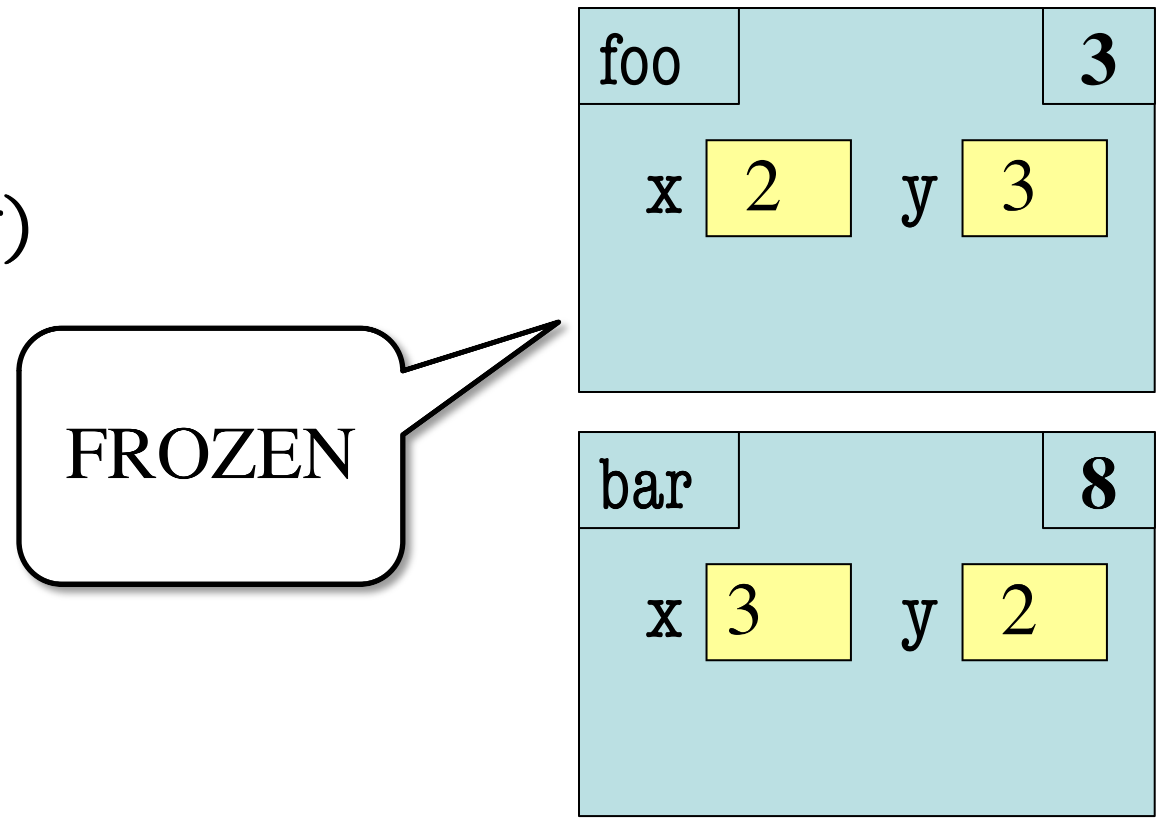
One Function Calling Another

```
def foo(x):
    y = x+1
    z = bar(y)
    return z

def bar(x):
    y = x-1
    return y

w = foo(2)
```



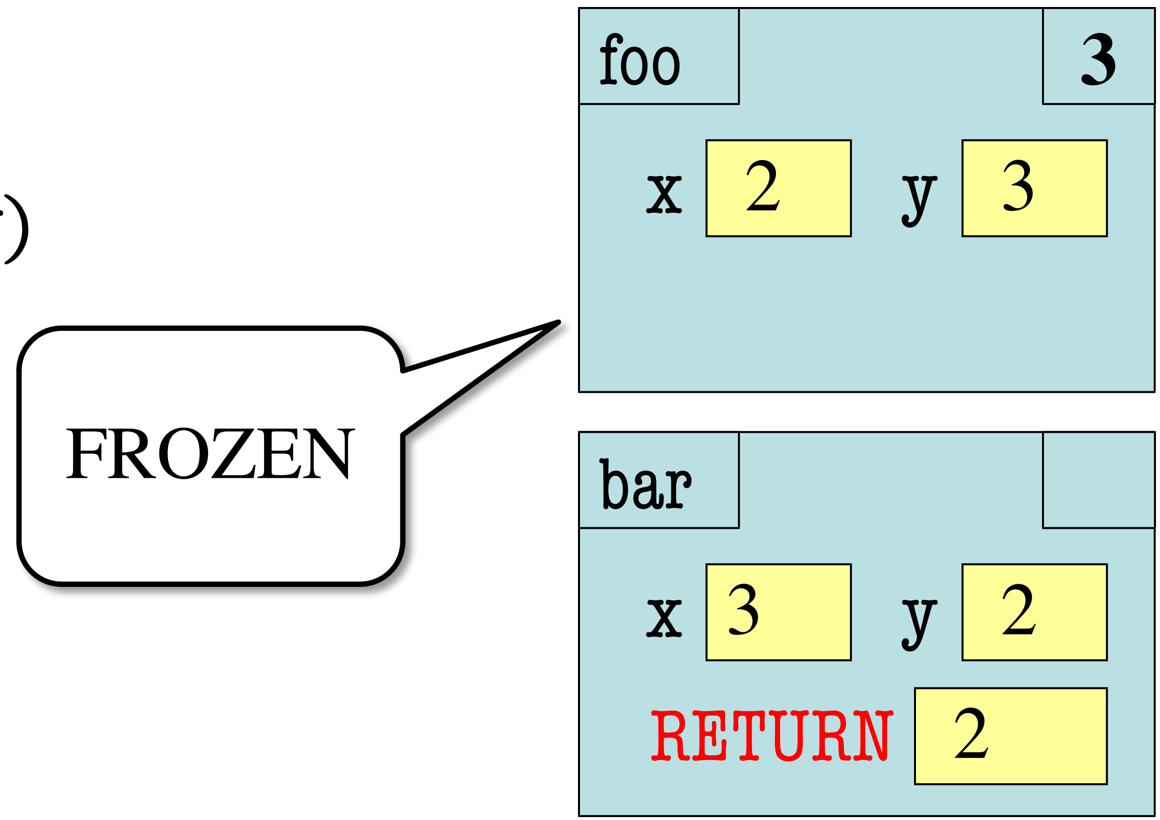
One Function Calling Another

```
1.  def foo(x):
2.      y = x+1
3.      z = bar(y)
4.      return z
5.
6.  def bar(x):
7.      y = x-1
8.      return y
9.
10. w = foo(2)
```



One Function Calling Another

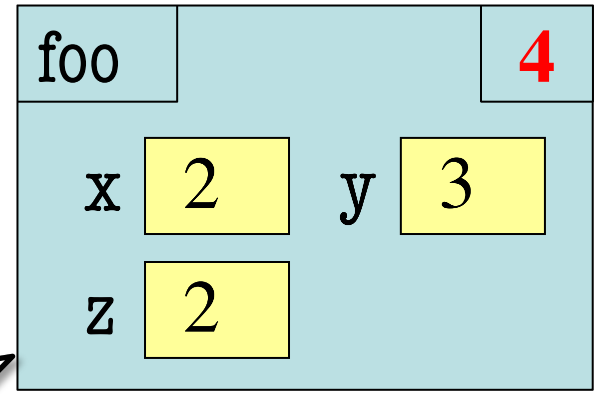
```
def foo(x):
    y = x+1
    z = bar(y)
    return z

def bar(x):
    y = x-1
    return y

w = foo(2)
```



One Function Calling Another

```
def foo(x):
    y = x+1
    z = bar(y)
    return z

def bar(x):
    y = x-1
    return y

w = foo(2)
```

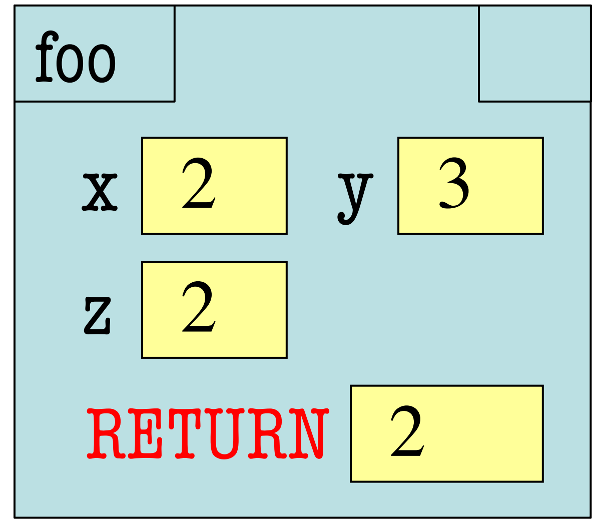
UNFREEZE



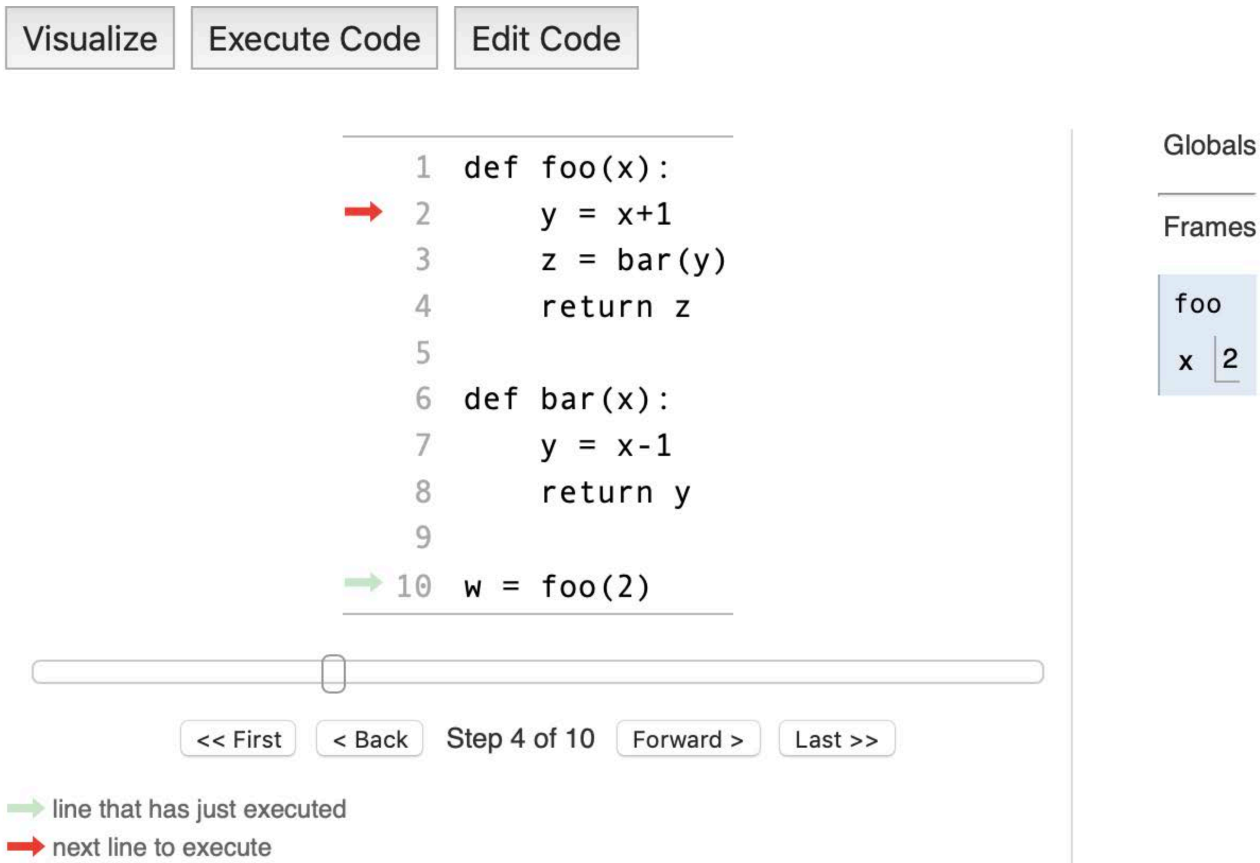
ERASE WHOLE FRAME

One Function Calling Another

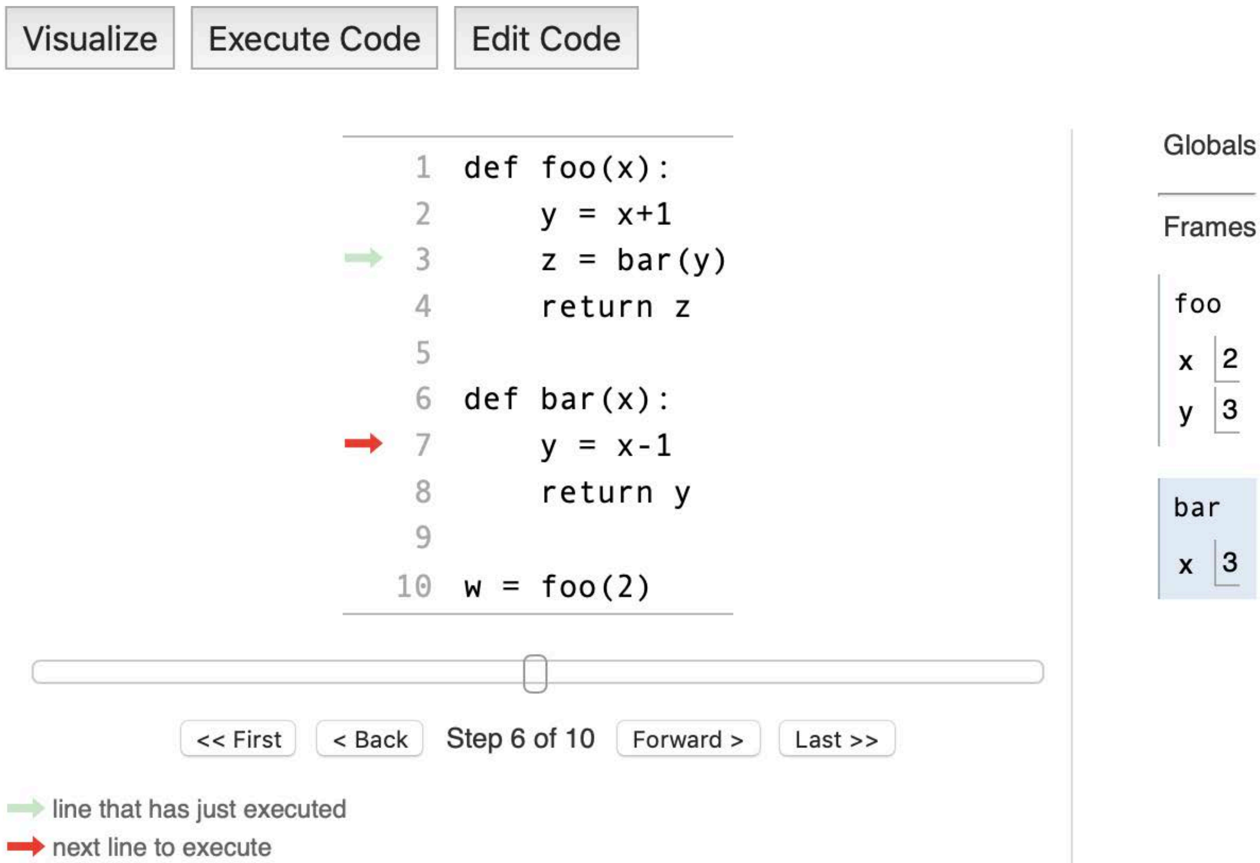
```
def foo(x):
    y = x+1
    z = bar(y)
    return z

def bar(x):
    y = x-1
    return y

w = foo(2)
```



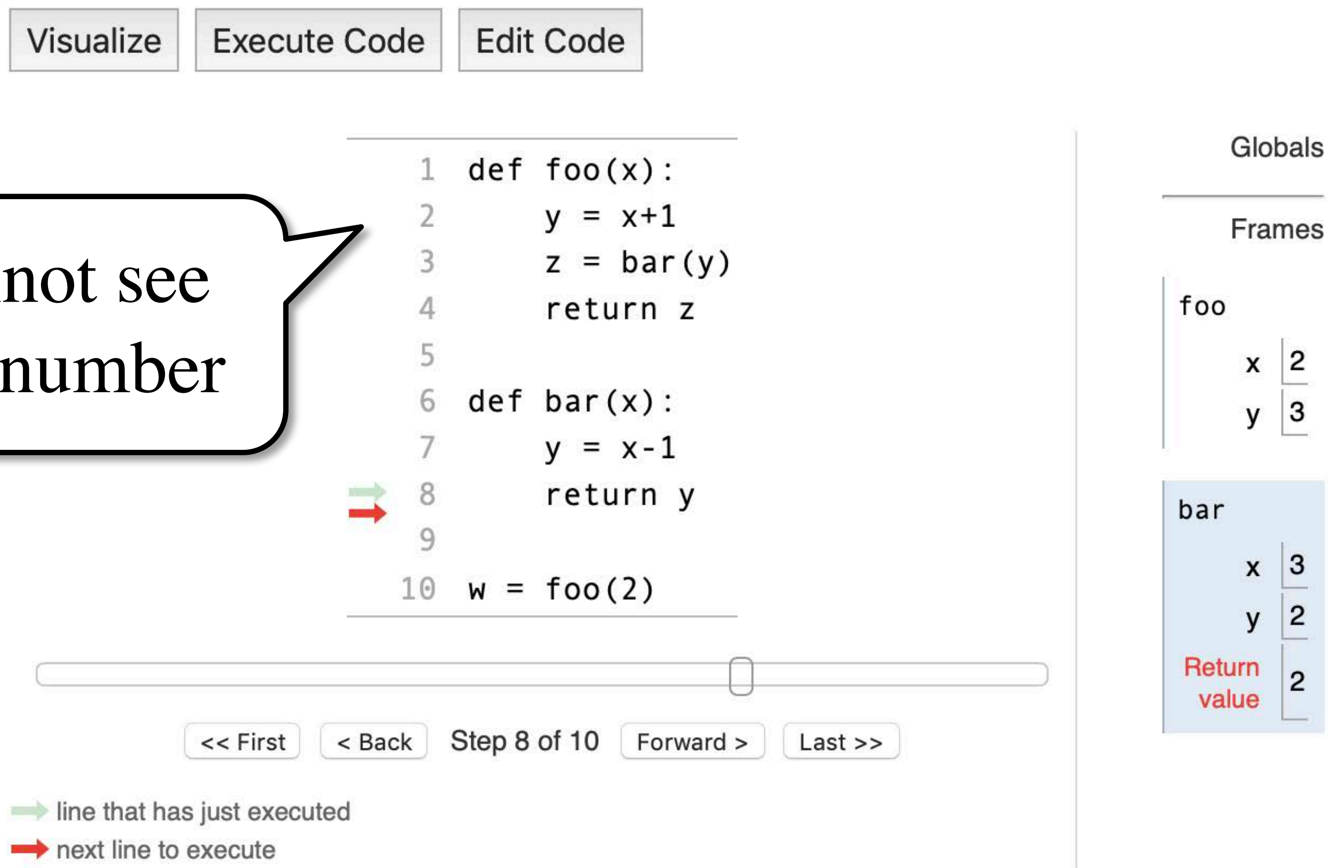
Viewing in the Python Tutor



Viewing in the Python Tutor



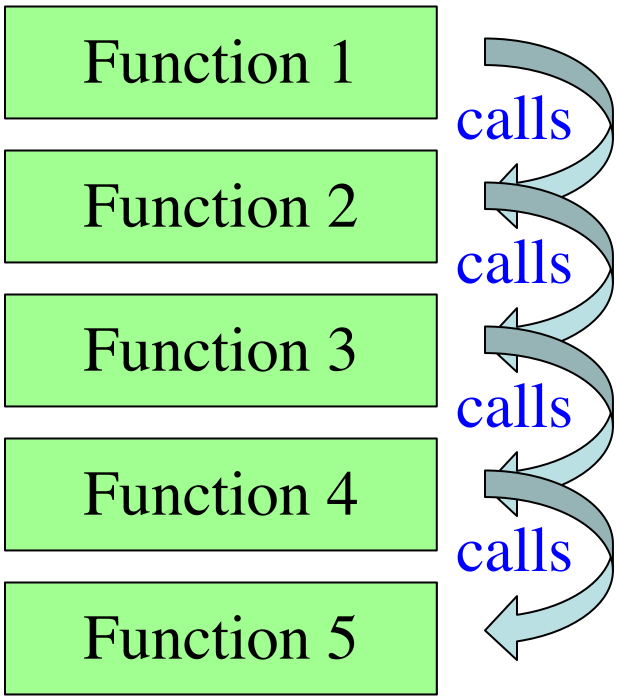
Viewing in the Python Tutor



Cannot see line number

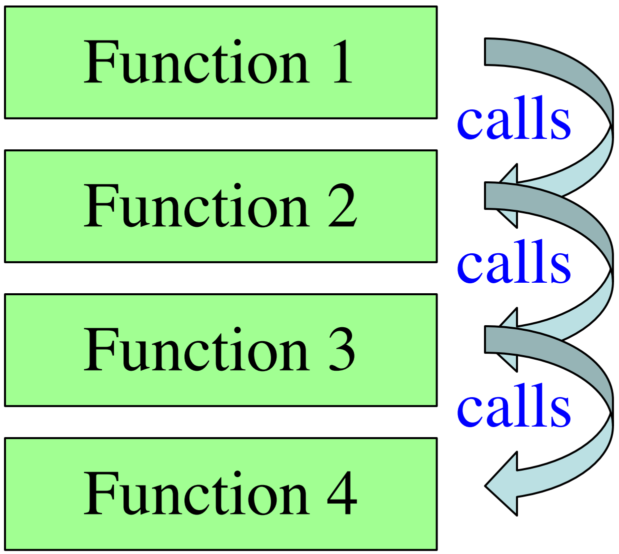
The Call Stack

- Functions are "stacked"
  - Cannot remove one above w/o removing one below
  - Sometimes draw bottom up (better fits the metaphor)
  - Top down because of Tutor
- Effects your memory
  - Need RAM for **entire stack**
  - An issue in adv. programs



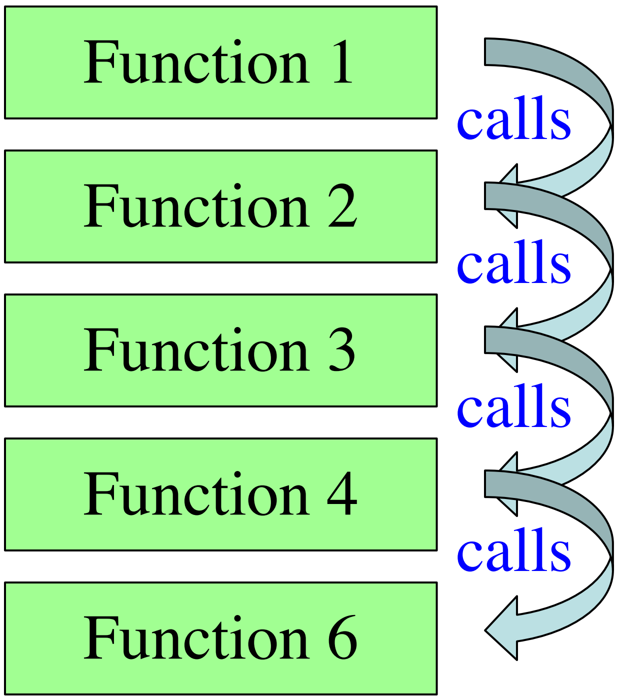
The Call Stack

- Functions are "stacked"
  - Cannot remove one above w/o removing one below
  - Sometimes draw bottom up (better fits the metaphor)
  - Top down because of Tutor
- Effects your memory
  - Need RAM for **entire stack**
  - An issue in adv. programs



The Call Stack

- Functions are "stacked"
  - Cannot remove one above w/o removing one below
  - Sometimes draw bottom up (better fits the metaphor)
  - Top down because of Tutor
- Effects your memory
  - Need RAM for **entire stack**
  - An issue in adv. programs



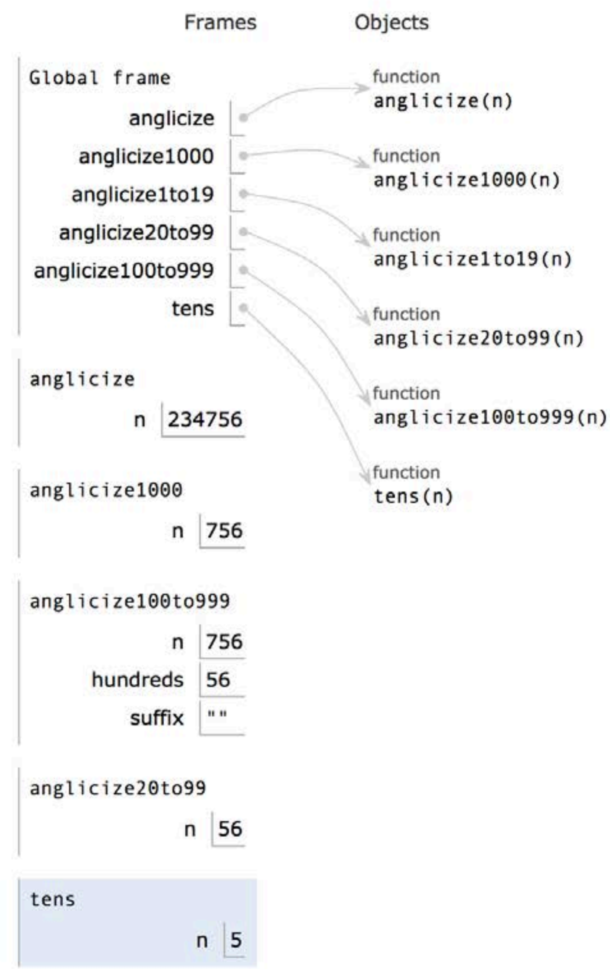
Anglicize Example

```python
def tens(n):
    """Returns: tens-word for n

    Parameter: the integer to anglicize
    Precondition: n in 2..9"""
    if n == 2:
        return 'twenty'
    elif n == 3:
        return 'thirty'
    elif n == 4:
        return 'forty'
    elif n == 5:
        return 'fifty'
    elif n == 6:
        return 'sixty'
    elif n == 7:
        return 'seventy'
    elif n == 8:
        return 'eighty'

    return 'ninety'
```

<< First | < Back | Step 26 of 89 | Forward > | Last >>

line that has just executed

next line to execute



Anglicize Example