

Lecture 10

Memory in Python

Announcements For This Lecture

Assignment 1

- Work on your revisions
 - Read feedback carefully
 - Partial credit after Friday
- Early survey results
 - 380 responded so far
 - Deadline is Friday
 - **Avg Time:** 6.6 hours
 - **STD Dev:** 3.6 hours

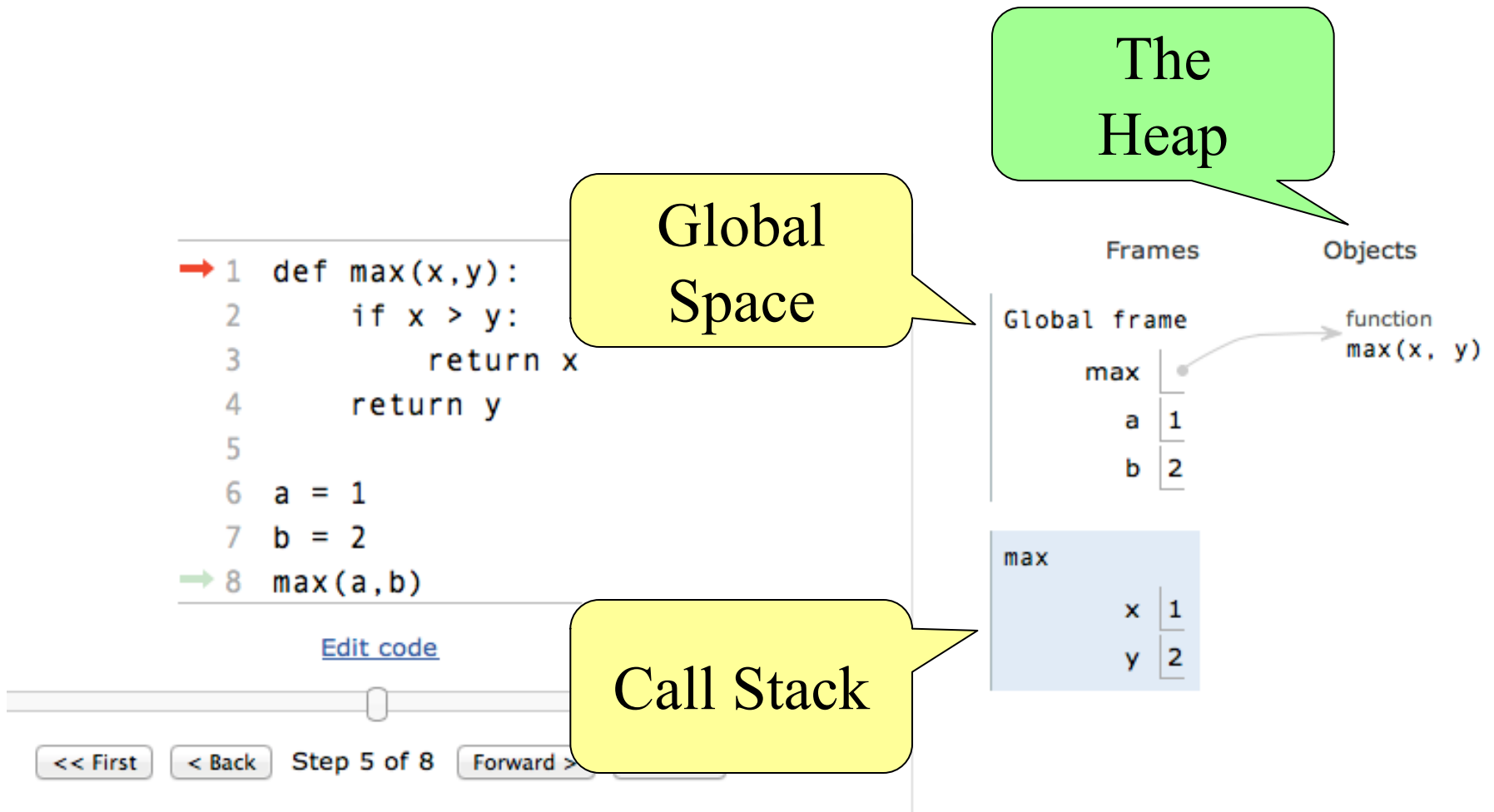
More Assignments

- Assignment 2 due **Sunday**
 - Scan and submit online
 - Upload before midnight
 - **Late:** -10% per day
 - No lates after Wednesday
- Assignment 3 up **Sunday**
 - Due Fri October 9
 - Should take as long as A1
 - Graded before exam

Speaking of the Exam

- **Prelim 1 is Oct 15th at 7:30-9:00**
 - Material is up to October 1st
 - Questions come from labs or assignments
- **How do you study for it?**
 - Will post a study guide this weekend
 - Can also look at old exams on web page
- **Conflict with Prelim time?**
 - Amy will post a Prelim 1 Conflict form
 - Do not submit if you have no conflict

The Three “Areas” of Memory



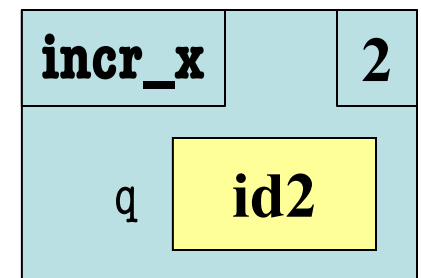
Global Space

- 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 & function definitions!**
 - Will see more on this in a bit

p id2

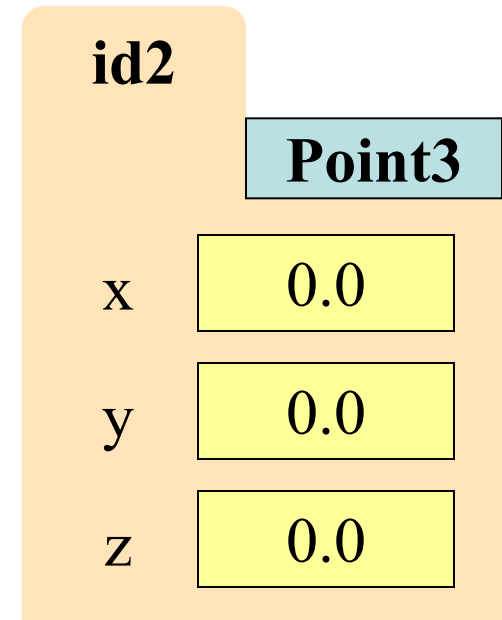
The Call Stack

- 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*



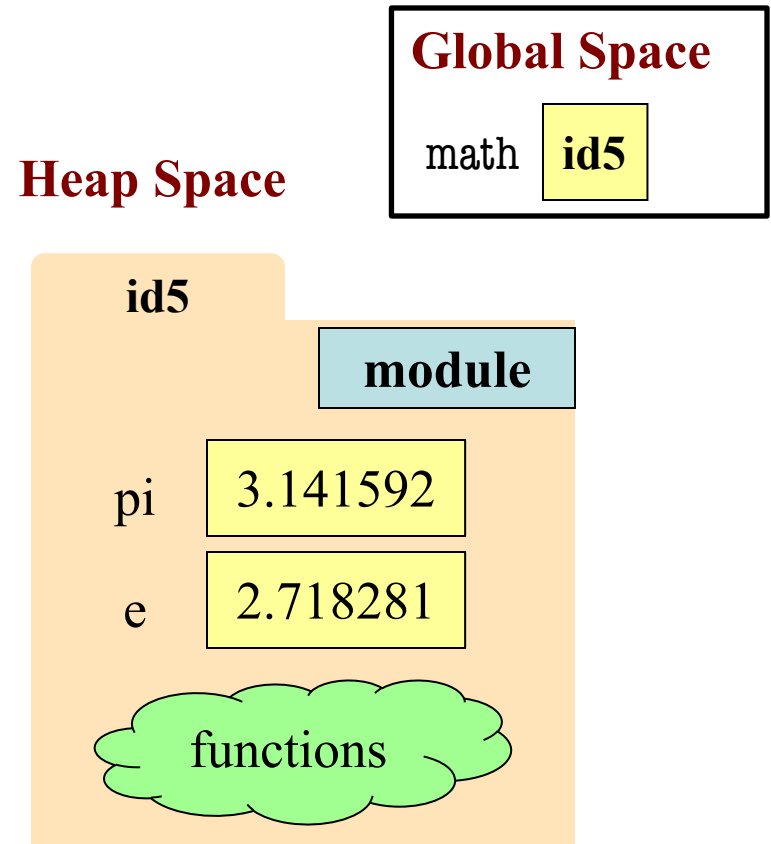
Everything is an Object!

- Last time we saw that everything is an object
 - Must have a folder in the heap
 - Must have variable in global space, call stack
 - But ignore basic types (int, float, bool, str)
- Includes **modules** and **function definitions**!
 - Object is created by import
 - Object is created by def
 - Already seen this in Python Tutor

Modules and Global Space

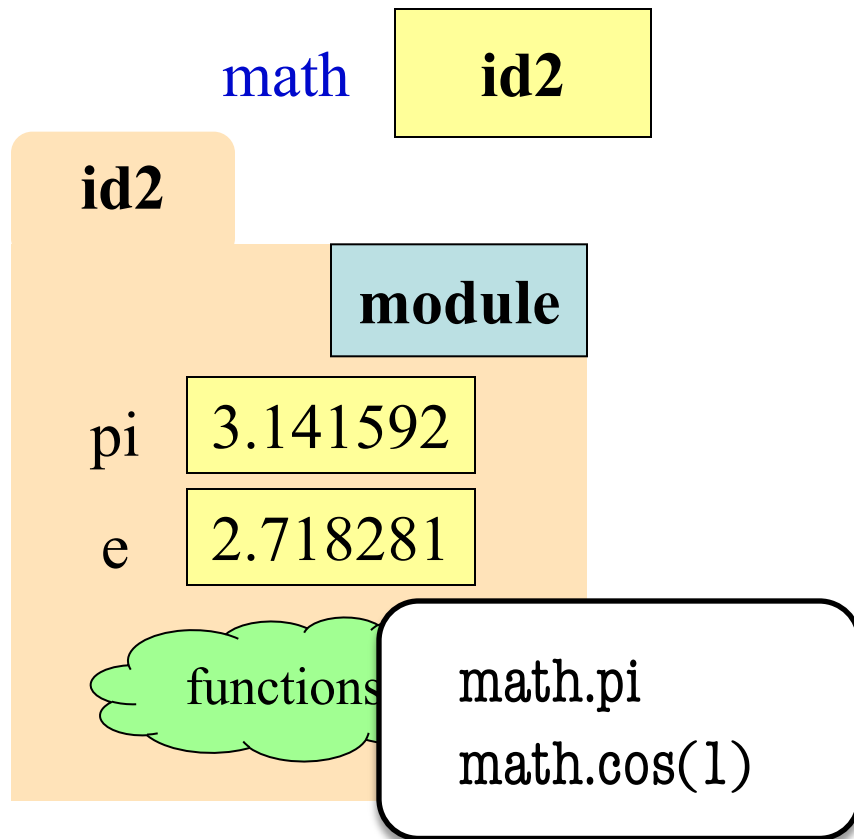
- Importing a module:
 - Creates a global variable (same name as module)
 - Puts contents in a **folder**
 - Module variables
 - Module functions
 - Puts folder id in variable
- **from** keyword dumps contents to global space

```
import math
```

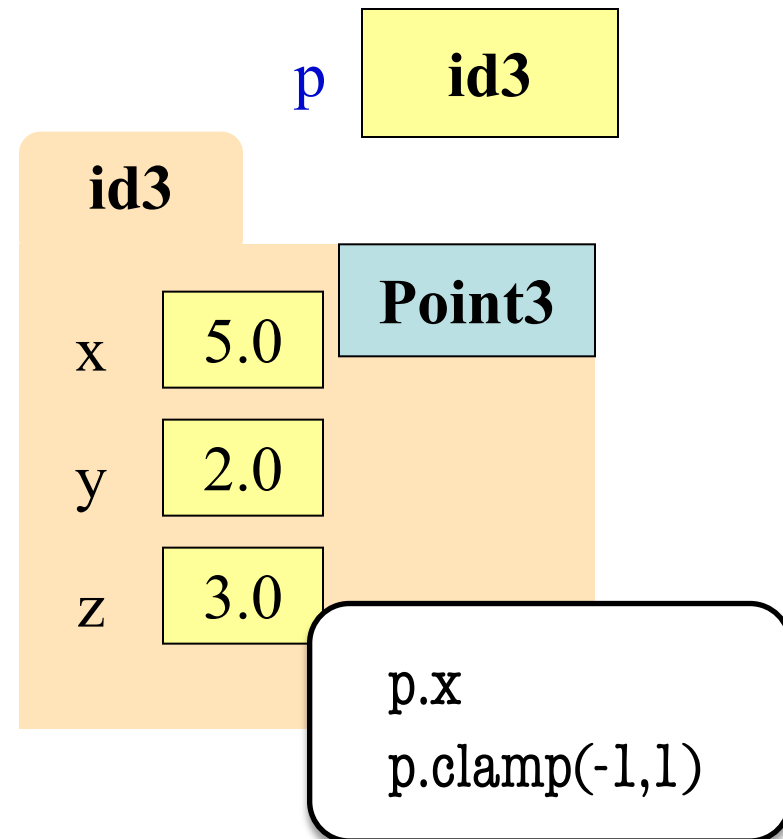


Modules vs Objects

Module



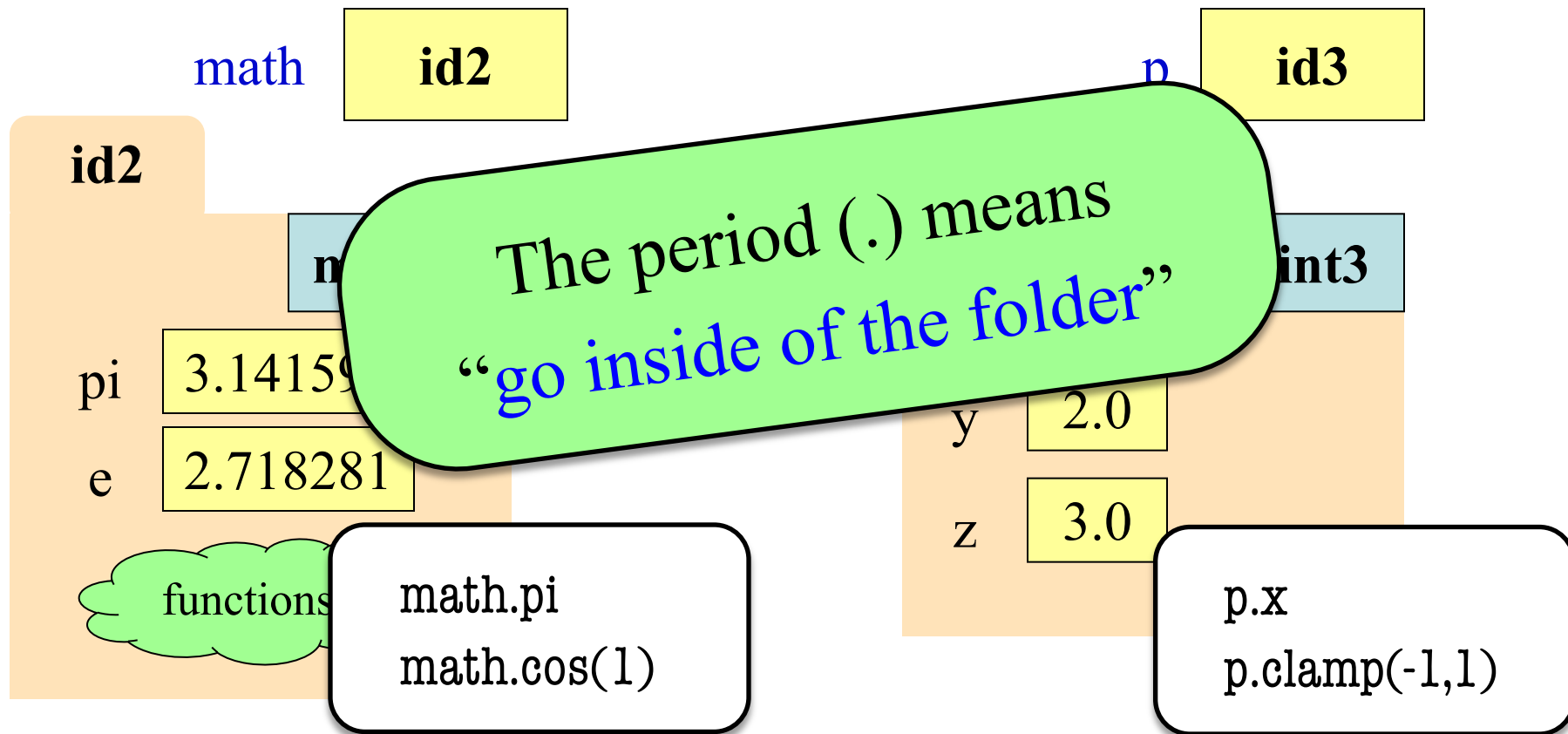
Object



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 instances**
 - Infinitely many
 - Reusable
- Each **instance**
 - Only one possibility for pi, cosine
 - That is why we import them
 - Sometimes refer to as *singleton* objects

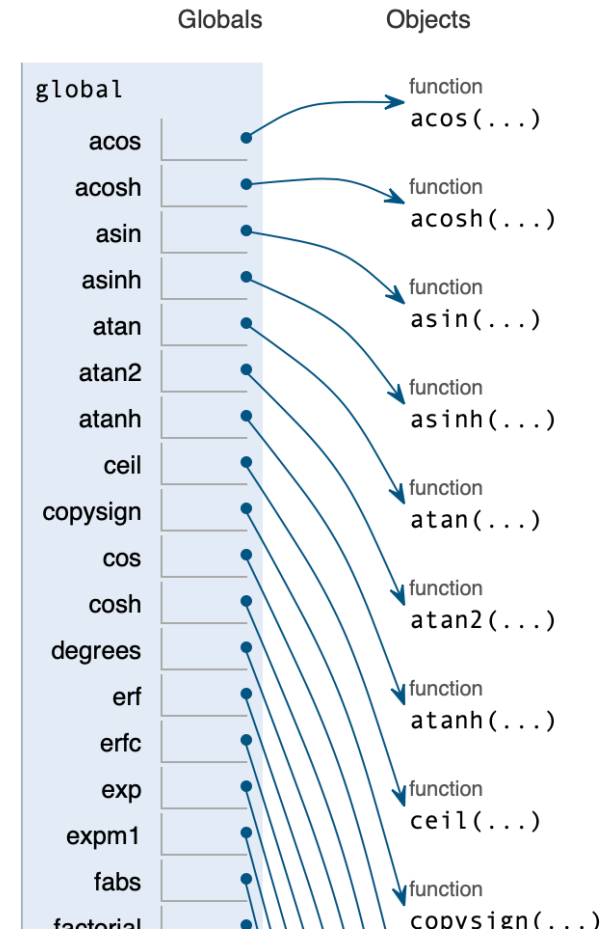
Choice is an advanced topic
beyond scope of this course

How About `import *`?

```
→ 1 from math import *  
→ 2 x = cos(1)
```

<< First < Back Step 2 of 2 Forward > Last >>

Ouch!



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

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

Body

Global Space

to_centrigrade id6

Heap Space

id6

function

Body

- Variable vs. Call

```
>>> to_centrigrade
```

```
<fun to_centrigrade at 0x100498de8>
```

```
>>> to_centrigrade (32)
```

```
0.0
```

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 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

```
1 def add_one(x):  
2     """Returns x+1"""  
→ 3     return x+1  
4  
5 y = add_one  
→ 6 z = y(2)
```

<< First

< Back

Step 4 of 5

Forward >

Last >>

Globals

Objects

global

add_one

y

function
add_one(x)

Frames

add_one

x

2

Frame remembers
the original name

Example: `add_one`



Usage is an advanced topic
beyond scope of this course

Frame remembers
the original name

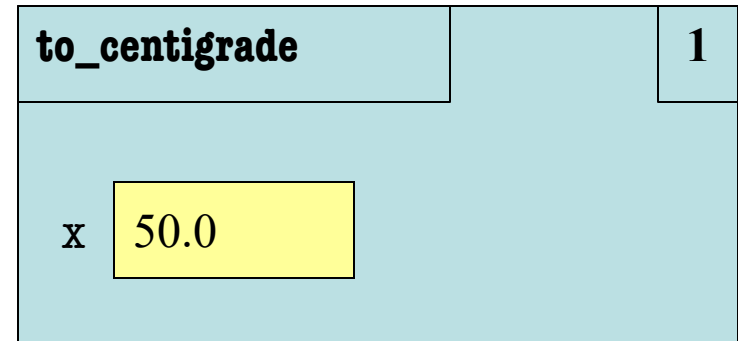
Why Show All This?

- 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
 - This means understanding the **call stack**

Recall: Call Frames

1. Draw a frame for the call
2. Assign the argument value to the parameter (in frame)
3. Execute the function body
 - Look for variables in the frame
 - If not there, look for global variables with that name
4. Erase the frame for the call

Call: to_centigrade(50.0)



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

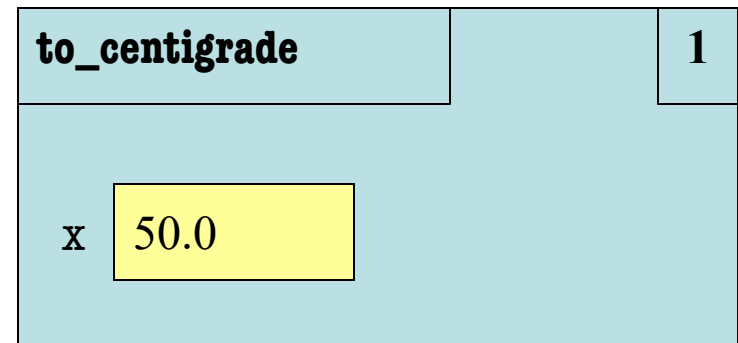
Aside: What Happens Each Frame Step?

- The instruction counter **always** changes
- The contents only **change** if
 - You add a new variable
 - You change an existing variable
 - You delete a variable
- If a variable refers to a **mutable object**
 - The contents of the folder might change

Recall: Call Frames

1. Draw a frame for the call
2. Assign the argument value to the parameter (in frame)
3. Execute the function body
 - Look for variables in the frame
 - If not there, look for global variables with that name
4. Erase the frame for the call

Call: to_centigrade(50.0)



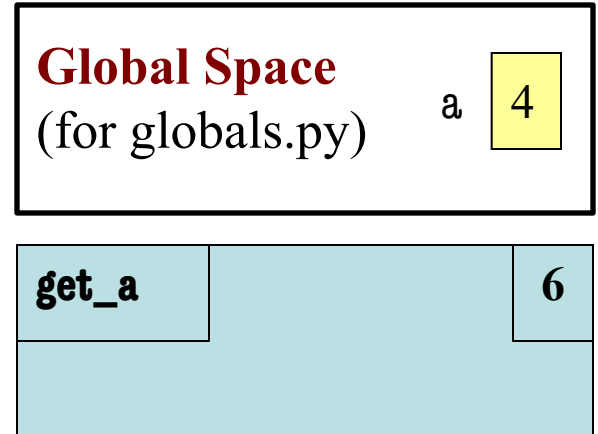
What is happening here?

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

Function Access to Global Space

- Consider code to right
 - Global variable `a`
 - Function definition `get_a`
- Consider the call `get_a()`
 - Call frame to the right
 - What happens?

A: It crashes
B: Returns None
C: Returns 4
D: I don't know



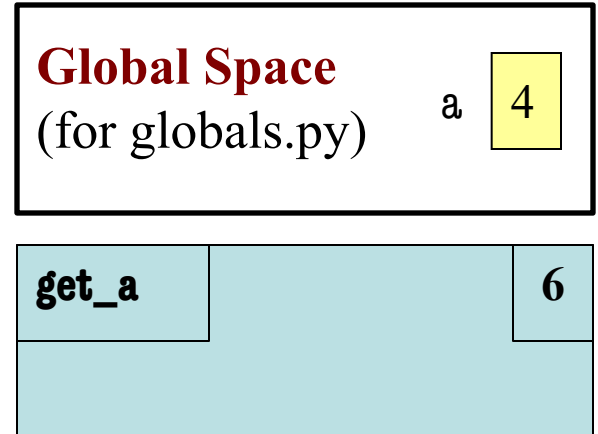
```
# globals.py
"""Show how globals work"""
a = 4 # global space

def get_a():
    return a
```

Function Access to Global Space

- Consider code to right
 - Global variable `a`
 - Function definition `get_a`
- Consider the call `get_a()`
 - Call frame to the right
 - What happens?

A: It crashes
B: Returns None
C: Returns 4 **CORRECT**
D: I don't know

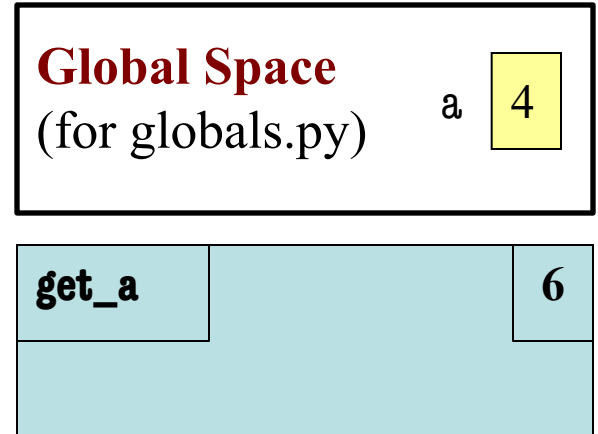


```
# globals.py
"""Show how globals work"""
a = 4 # global space

def get_a():
    |    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`
 - `temperature.to_centigrade` uses global for `temperature`
- But **cannot** change values
 - Makes a *new local variable*!
 - Why we limit to constants

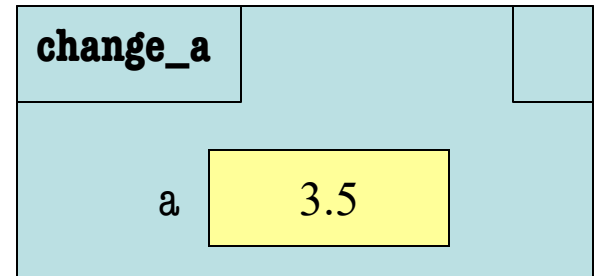


```
# globals.py
"""Show how globals work"""
a = 4 # global space

def get_a():
    |     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`
 - `temperature.to_centigrade` uses global for `temperature`
- But **cannot** change values
 - Makes a *new local variable*!
 - Why we limit to constants



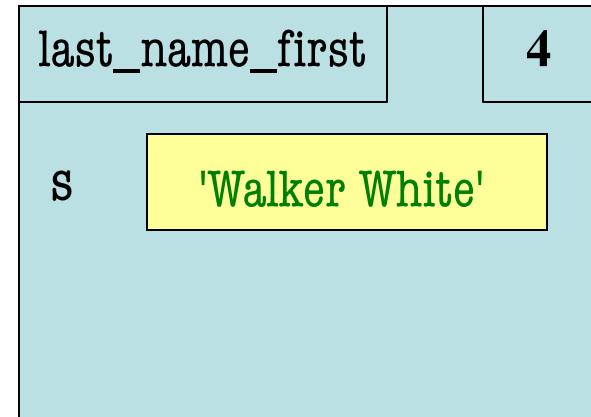
```
# globals.py
"""Show how globals work"""
a = 4 # global space

def change_a():
    | a = 3.5 # local variable
```

Frames and Helper Functions

```
1. def last_name_first(s):
2.     """Precond: s in the form
3.     'first-name last-name' """
4.     first = first_name(s)
5.     last = last_name(s)
6.     return last + ',' + first
7.
8. def first_name(s):
9.     """Precond: see above"""
10.    end = s.find(' ')
11.    return s[0:end]
```

Call: last_name_first('Walker White'):

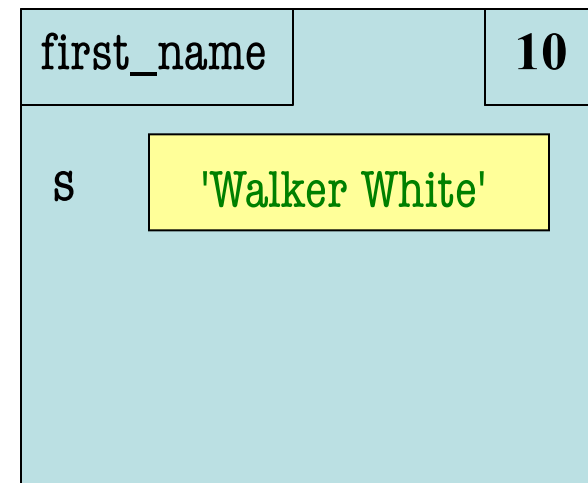
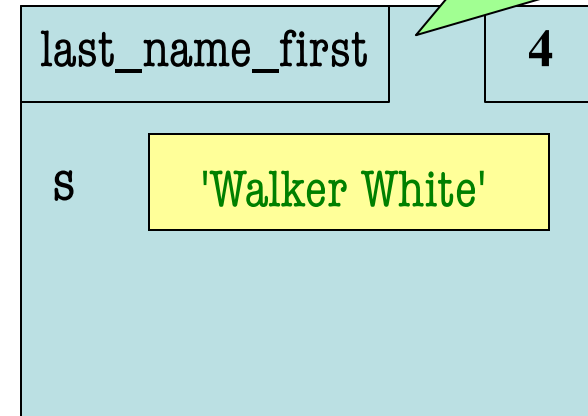


Frames and Helper Functions

```
1. def last_name_first(s):
2.     """Precond: s in the form
3.     'first-name last-name' """
4.     first = first_name(s)
5.     last = last_name(s)
6.     return last + ',' + first
7.
8. def first_name(s):
9.     """Precond: see above"""
10.    end = s.find(' ')
11.    return s[0:end]
```

Call: last_name_first('Walker White')

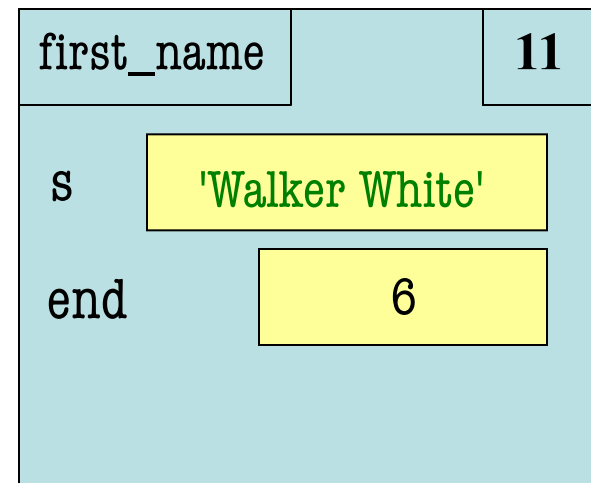
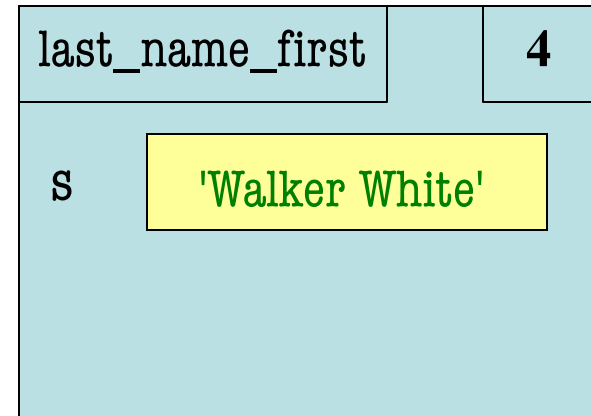
Not done. Do not erase!



Frames and Helper Functions

```
1. def last_name_first(s):
2.     """Precond: s in the form
3.     'first-name last-name' """
4.     first = first_name(s)
5.     last = last_name(s)
6.     return last + ',' + first
7.
8. def first_name(s):
9.     """Precond: see above"""
10.    end = s.find(' ')
11.    return s[0:end]
```

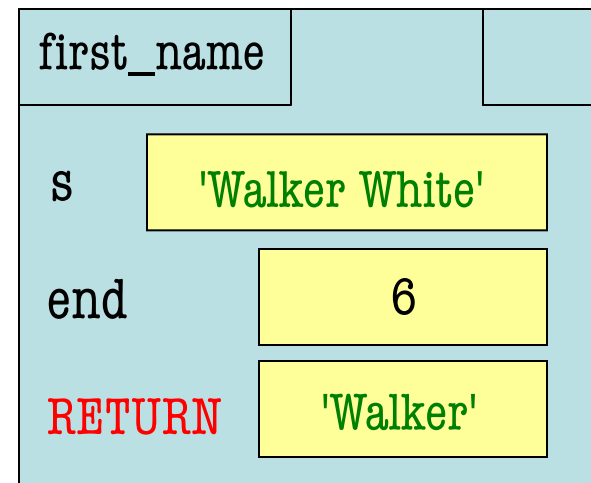
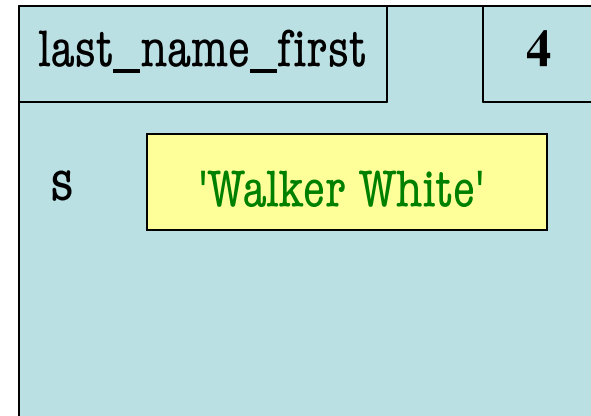
Call: last_name_first('Walker White'):



Frames and Helper Functions

```
1. def last_name_first(s):
2.     """Precond: s in the form
3.     'first-name last-name' """
4.     first = first_name(s)
5.     last = last_name(s)
6.     return last + ',' + first
7.
8. def first_name(s):
9.     """Precond: see above"""
10.    end = s.find(' ')
11.    return s[0:end]
```

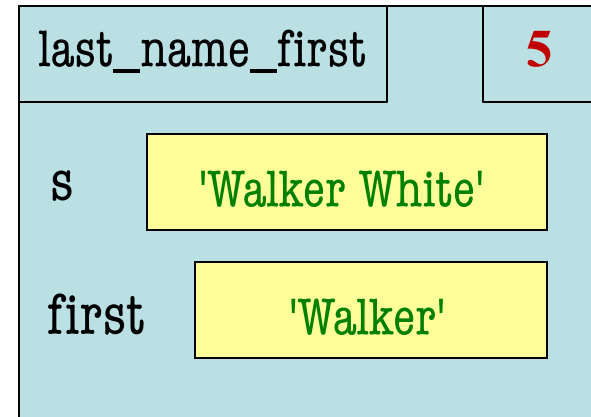
Call: last_name_first('Walker White'):



Frames and Helper Functions

```
1. def last_name_first(s):
2.     """Precond: s in the form
3.     'first-name last-name' """
4.     first = first_name(s)
5.     last = last_name(s)
6.     return last + ',' + first
7.
8. def first_name(s):
9.     """Precond: see above"""
10.    end = s.find(' ')
11.    return s[0:end]
```

Call: last_name_first('Walker White'):



ERASE WHOLE FRAME

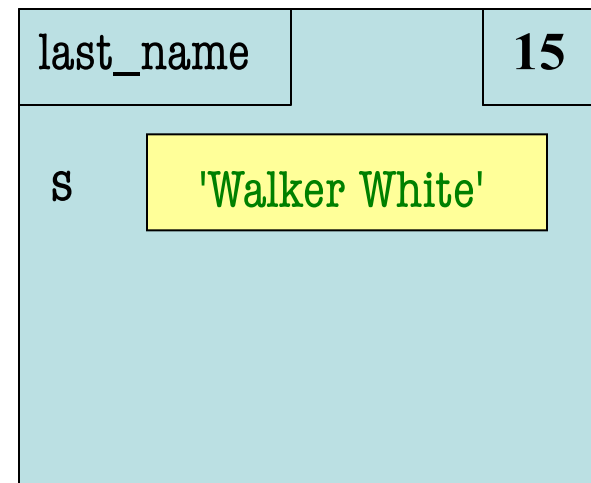
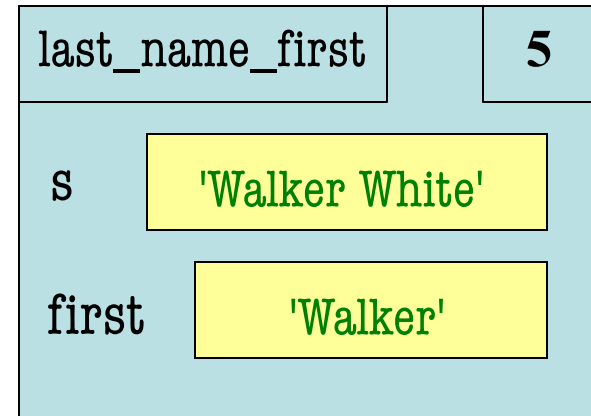
Frames and Helper Functions

```
1. def last_name_first(s):  
2.     """Precond: s in the form  
3.     'first-name last-name' """  
4.     first = first_name(s)  
5.     last = last_name(s)  
6.     return last + ',' + first
```

...

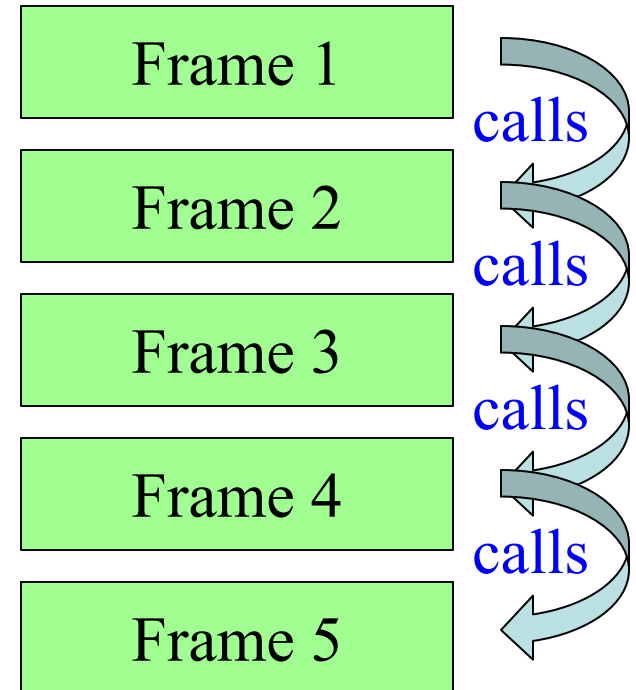
```
13. def last_name(s):  
14.     """Precond: see above"""  
15.     end = s.rfind(' ')  
16.     return s[end+1:]
```

Call: last_name_first('Walker White'):



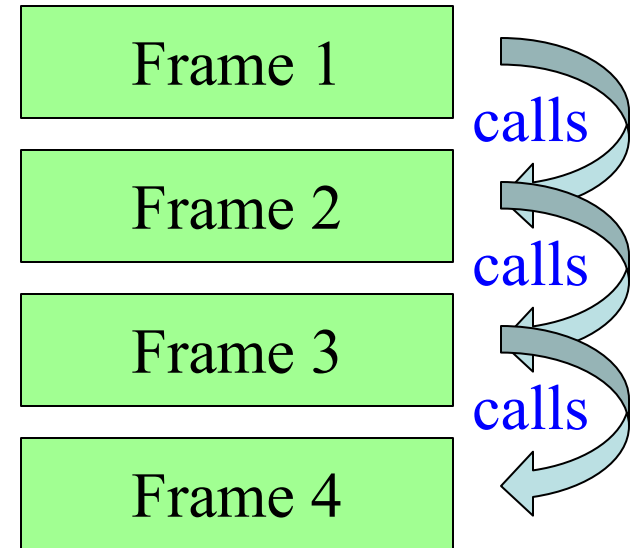
The Call Stack

- Functions are **stacked**
 - Cannot remove one above w/o removing one below
 - Sometimes draw bottom up (better fits the metaphor)
- Stack represents memory as a ***high water mark***
 - Must have enough to keep the **entire stack in memory**
 - Error if cannot hold stack



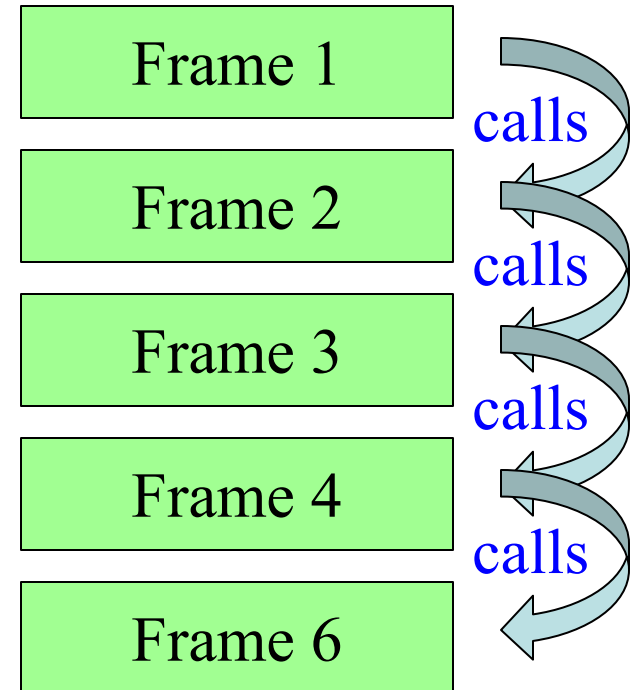
The Call Stack

- Functions are **stacked**
 - Cannot remove one above w/o removing one below
 - Sometimes draw bottom up (better fits the metaphor)
- Stack represents memory as a ***high water mark***
 - Must have enough to keep the **entire stack in memory**
 - Error if cannot hold stack



The Call Stack

- Functions are **stacked**
 - Cannot remove one above w/o removing one below
 - Sometimes draw bottom up (better fits the metaphor)
- Stack represents memory as a ***high water mark***
 - Must have enough to keep the **entire stack in memory**
 - Error if cannot hold stack



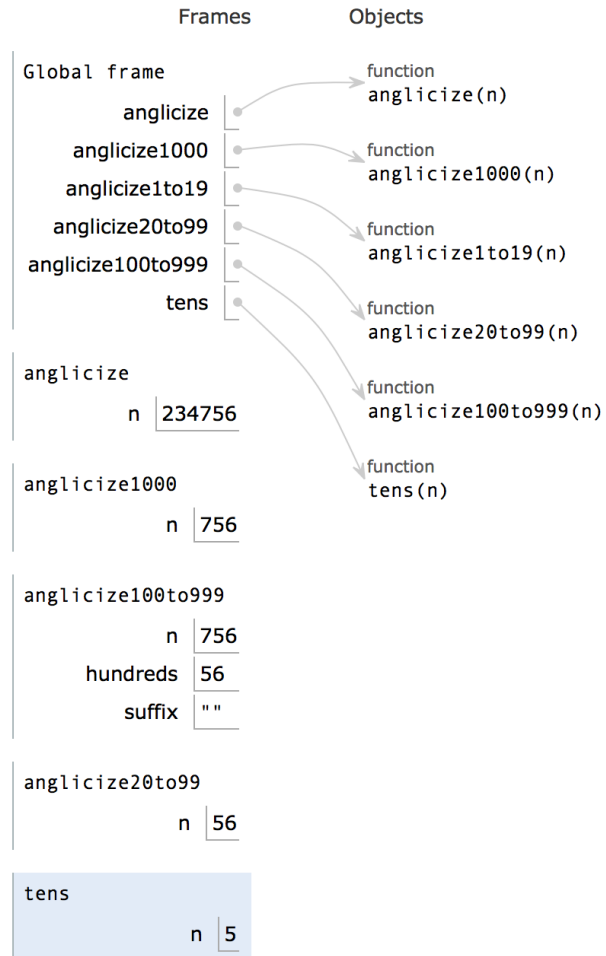
Anglicize Example

```
120
→ 121 def tens(n):
122     """Returns: tens-word for n
123
124     Parameter: the integer to anglicize
125     Precondition: n in 2..9"""
→ 126     if n == 2:
127         return 'twenty'
128     elif n == 3:
129         return 'thirty'
130     elif n == 4:
131         return 'forty'
132     elif n == 5:
133         return 'fifty'
134     elif n == 6:
135         return 'sixty'
136     elif n == 7:
137         return 'seventy'
138     elif n == 8:
139         return 'eighty'
140
141     return 'ninety'
142
```

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

→ line that has just executed

→ next line to execute



Anglicize Example

```
120
→ 121 def tens(n):
122     """Returns: tens-word for n
123
124     Parameter: the integer to anglicize
125     Precondition: n in 2..9"""
→ 126     if n == 2:
127         return 'twenty'
128     elif n == 3:
129         return 'thirty'
130     elif n == 4:
131         return 'forty'
132     elif n == 5:
133         return 'fifty'
134     elif n == 6:
135         return 'sixty'
136     elif n == 7:
137         return 'seventy'
138     elif n == 8:
139         return 'eighty'
140
141     return 'ninety'
142
```

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

→ line that has just executed

→ next line to execute

