

Lecture 10

Memory in Python

Announcements For This Lecture

Reading

- Reread all of Chapter 3



Assignments

- Work on your revisions
 - Want done by Sunday
- **Survey**: 501 responded
 - Remaining do by tomorrow
 - **Avg Time**: 7.3 hours
 - **STD Dev**: 3.6 hours
- Assignment 2 also Sunday
 - Scan and submit online
- Assignment 3 up Monday

Modeling Storage in Python

- **Global Space**

- What you “start with”
- Stores global variables
- Also **modules & functions!**
- Lasts until you quit Python

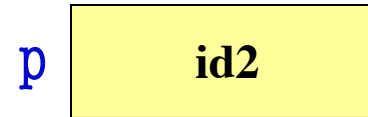
- **Call Frame**

- Variables in function call
- Deleted when call done

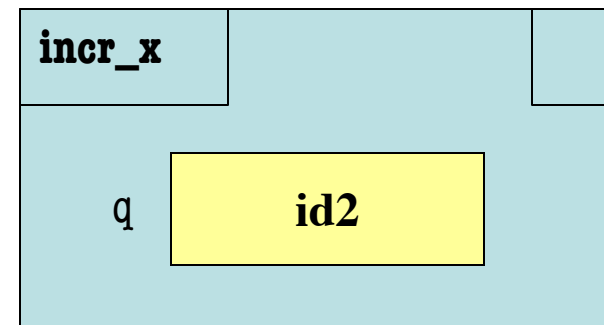
- **Heap Space**

- Where “folders” are stored
- Have to access indirectly

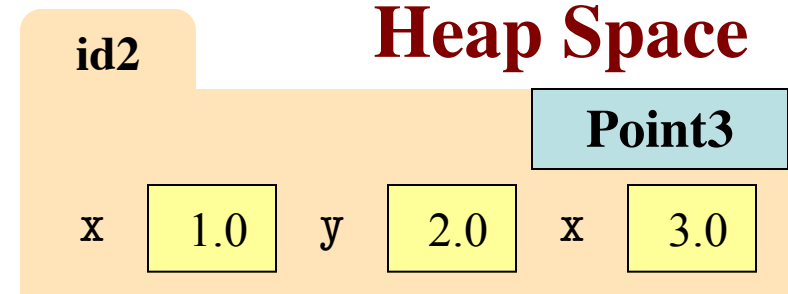
Global Space



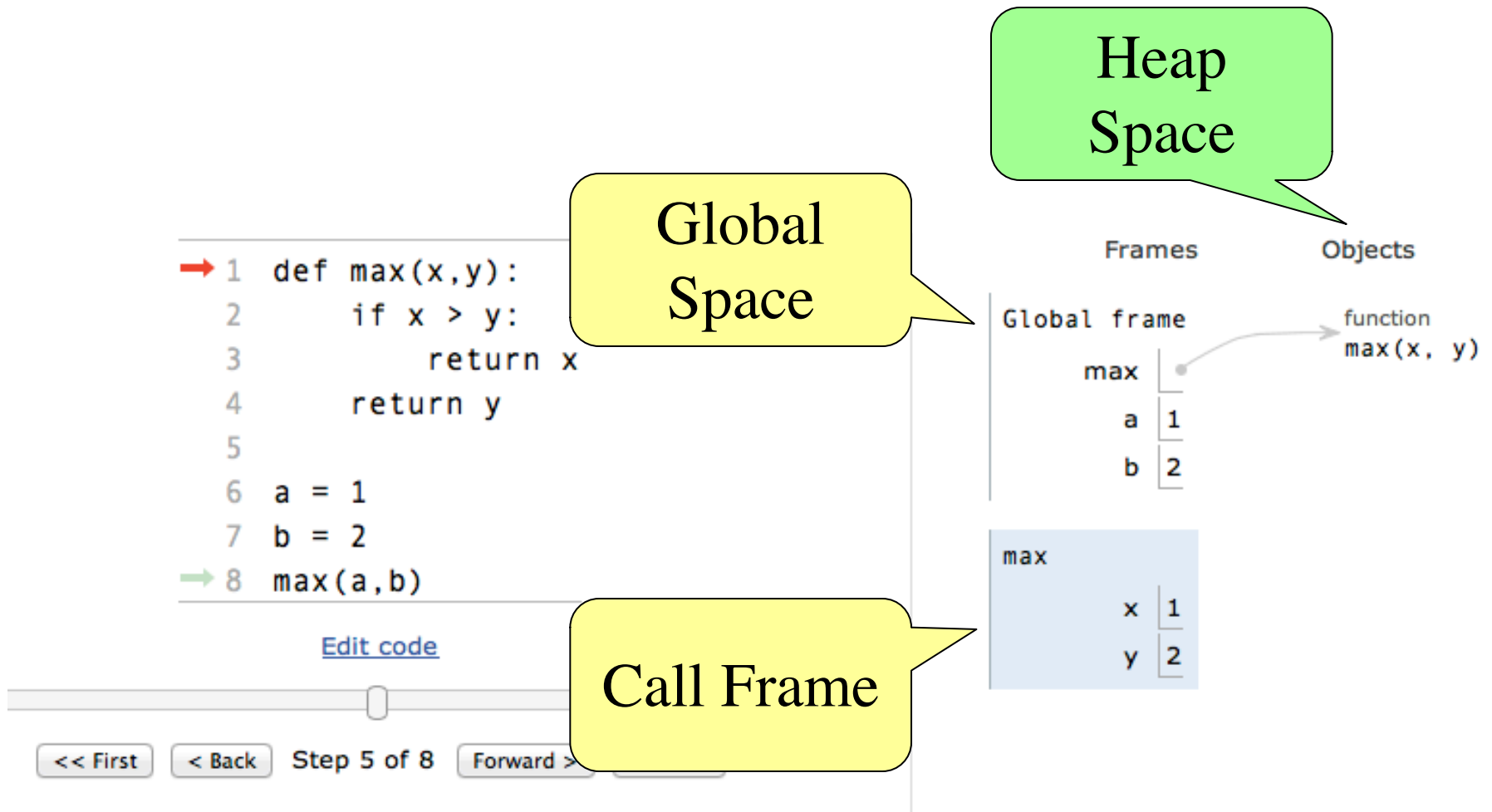
Call Frame



Heap Space



Memory and the Python Tutor



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**

- Variable vs. Call

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

Heap Space

id6

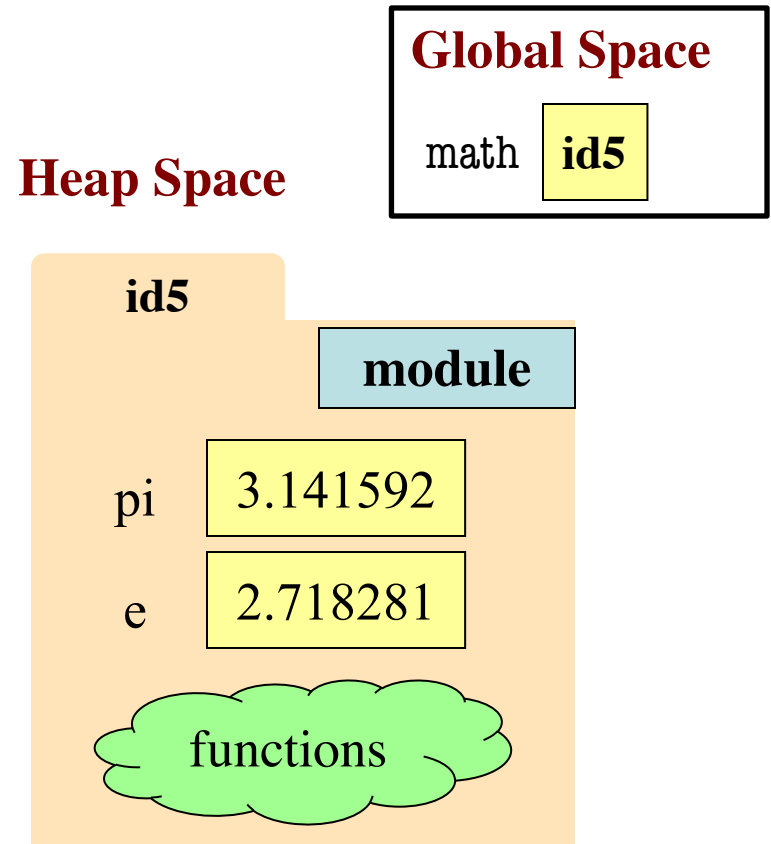
function

Body

Modules and Global Space

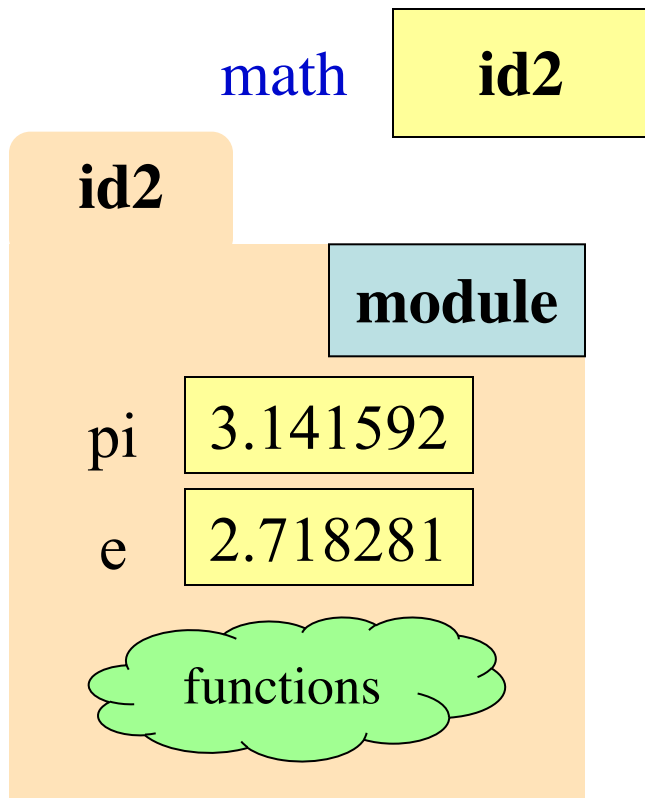
- Importing a module:
 - Creates a global variable (same name as module)
 - Puts contents in a **folder**
 - Module variables
 - Module functions
 - Puts folder id in variable
- **from** keyword dumps contents to global space

```
import math
```

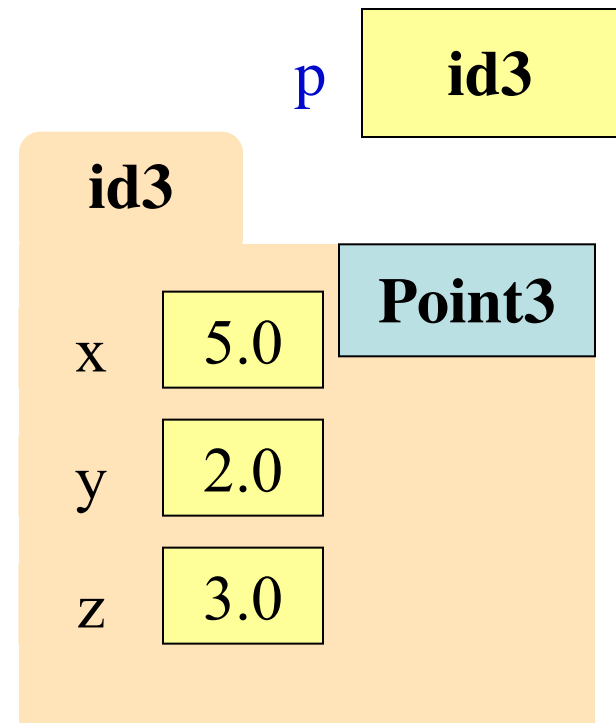


Modules vs Objects

Module

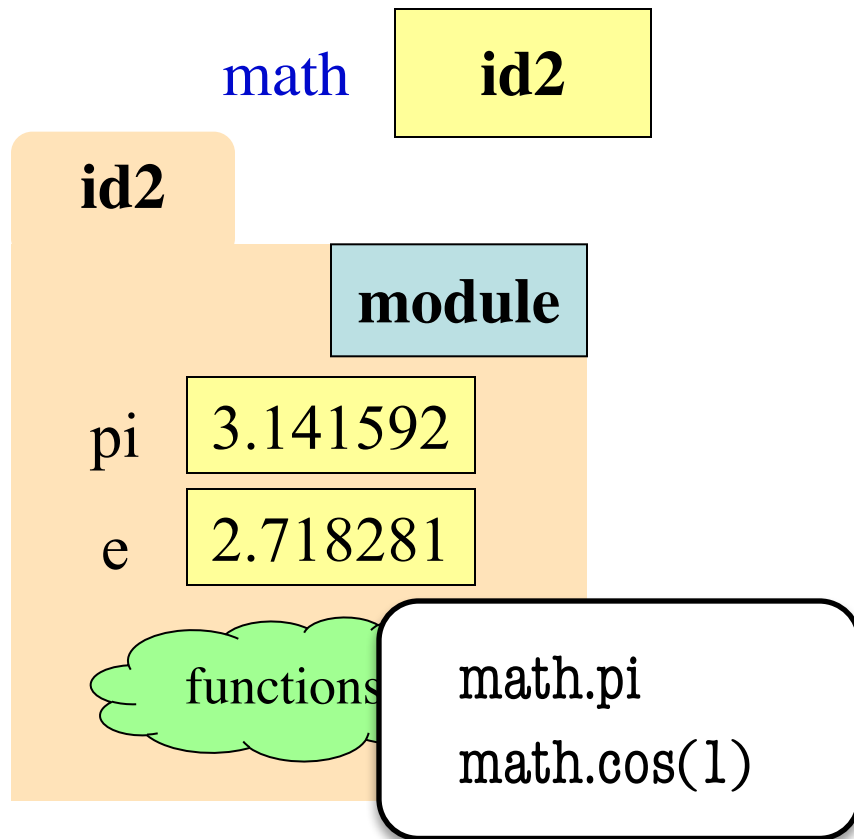


Object

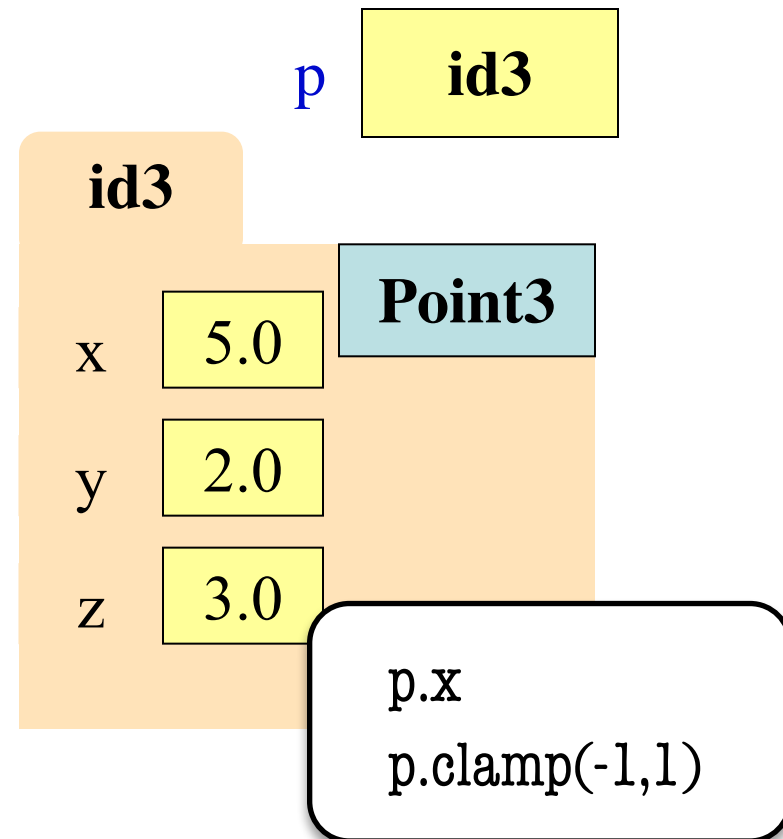


Modules vs Objects

Module



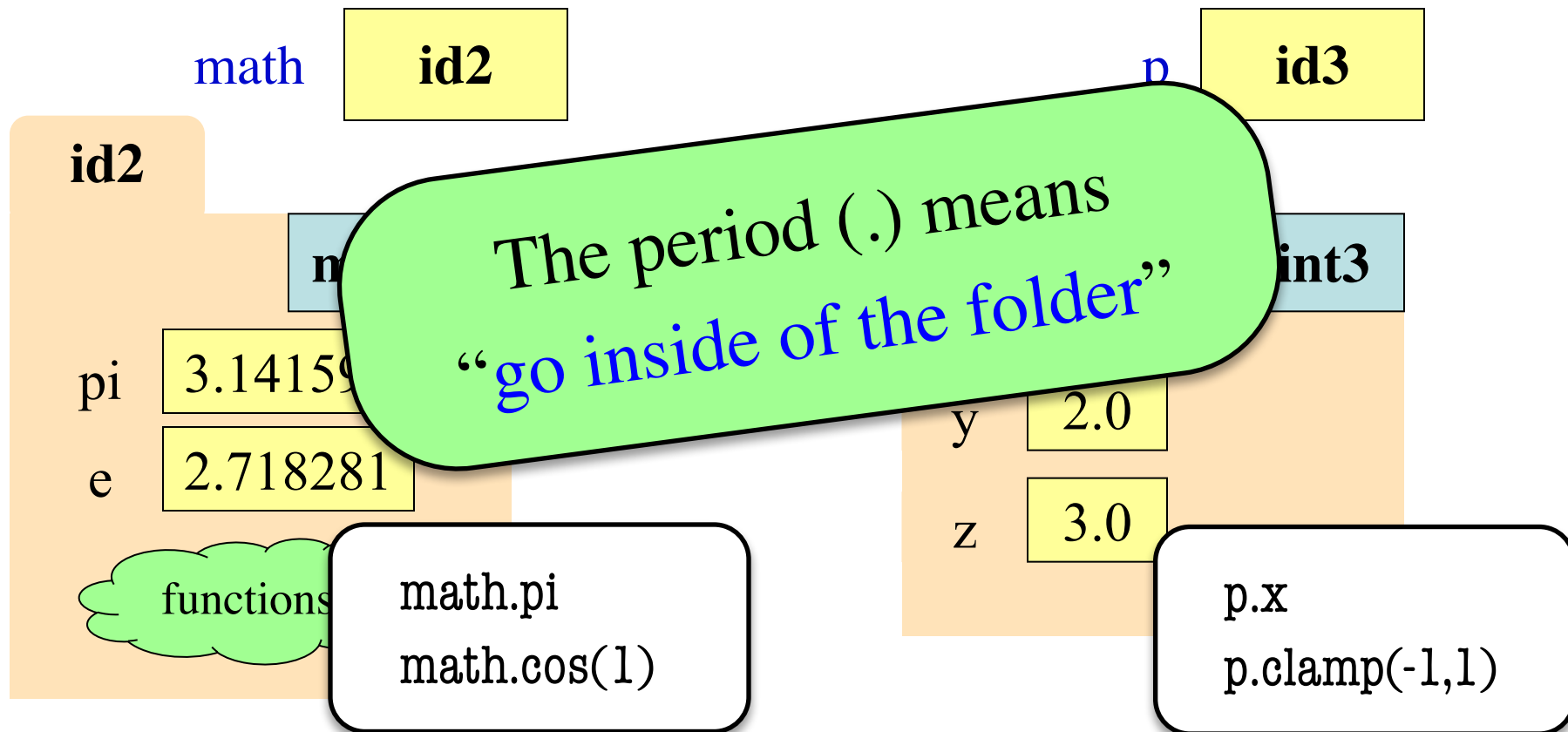
Object



Modules vs Objects

Module

Object

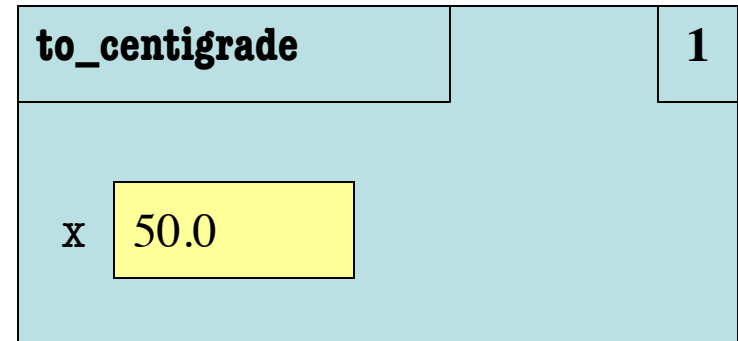


Recall: Call Frames

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?

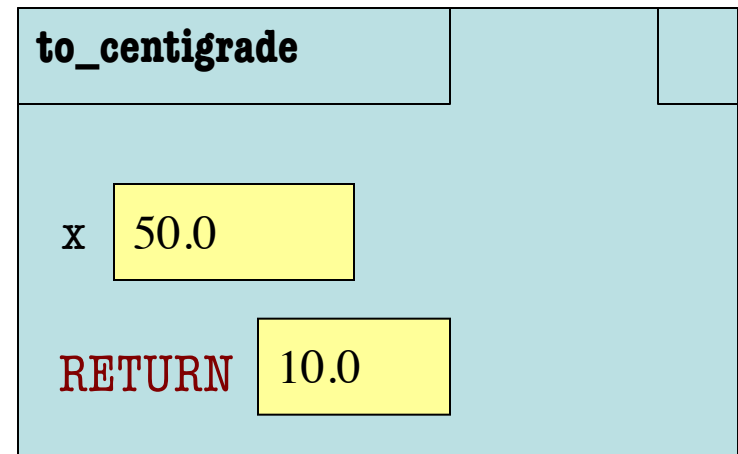
Only at the End!

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

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
```

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)

ERASE WHOLE FRAME

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

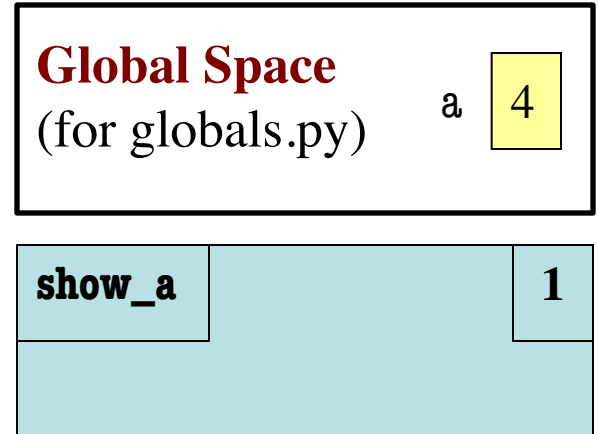
But don't actually
erase on an exam

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

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
 - Assignment to a global makes a new local variable!
 - Why we limit to constants

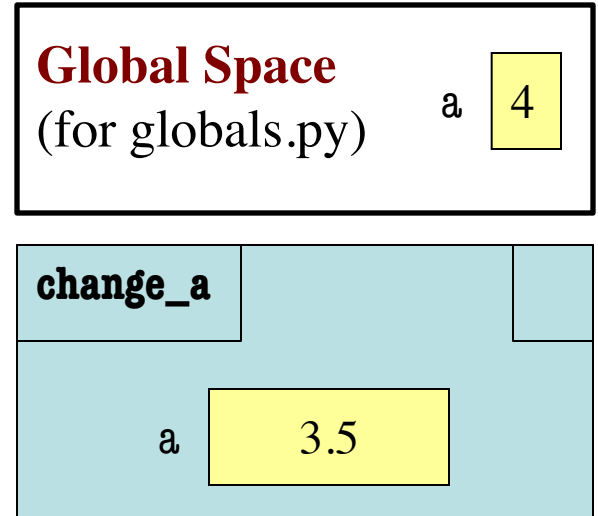


```
# globals.py
"""Show how globals work"""
a = 4 # global space

def show_a():
    print(a) # shows global
```

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
 - Assignment to a global 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
```

Call Frames and Objects

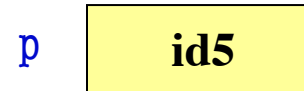
- Mutable objects can be altered in a function call
 - Object vars hold names!
 - Folder accessed by both global var & parameter
- **Example:**

```
def incr_x(q):  
1 |   q.x = q.x + 1
```

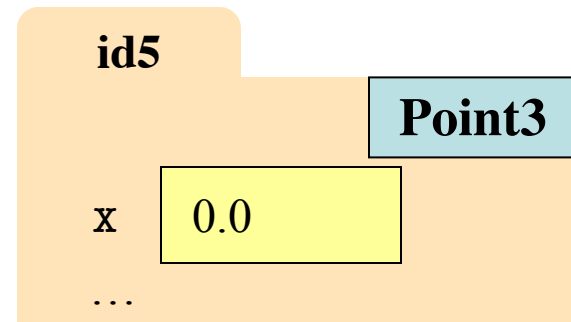
```
>>> p = Point3(0,0,0)
```

```
>>> incr_x(p)
```

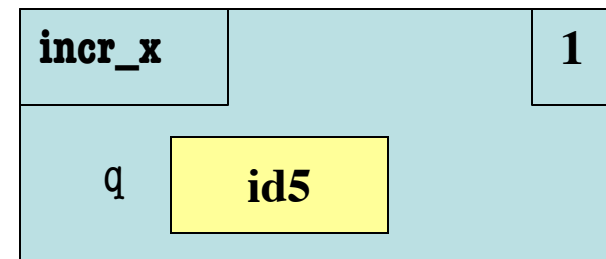
Global Space



Heap Space



Call Frame



Call Frames and Objects

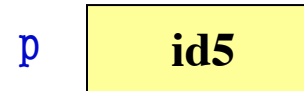
- Mutable objects can be altered in a function call
 - Object vars hold names!
 - Folder accessed by both global var & parameter
- **Example:**

```
def incr_x(q):  
1 |   q.x = q.x + 1
```

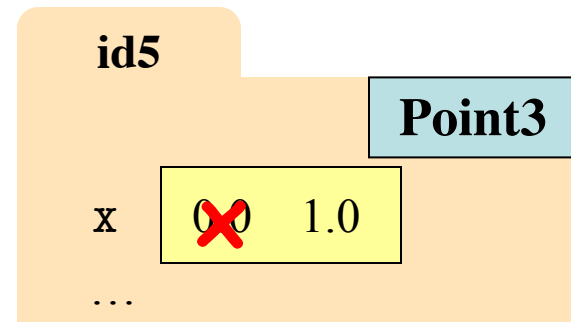
```
>>> p = Point3(0,0,0)
```

```
>>> incr_x(p)
```

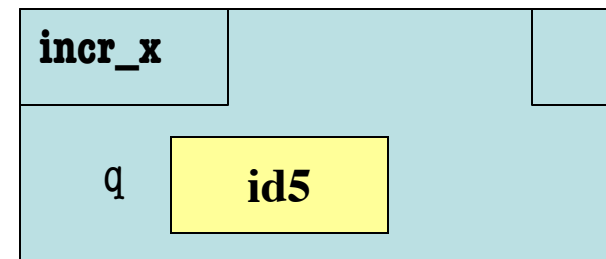
Global Space



Heap Space



Call Frame



Call Frames and Objects

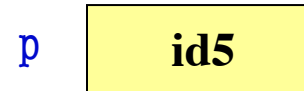
- Mutable objects can be altered in a function call
 - Object vars hold names!
 - Folder accessed by both global var & parameter
- **Example:**

```
def incr_x(q):  
1 |   q.x = q.x + 1
```

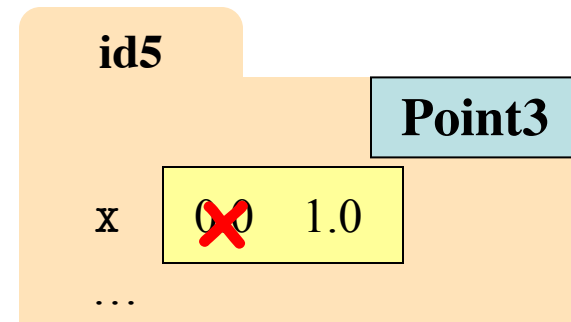
```
>>> p = Point3(0,0,0)
```

```
>>> incr_x(p)
```

Global Space



Heap Space



Call Frame

ERASE FRAME

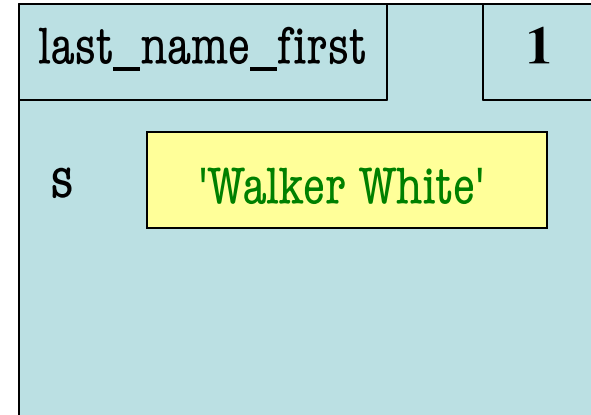
Frames and Helper Functions

def last_name_first(s):

"""**Precondition:** s in the form
<first-name> <last-name>"""

```
1 first = first_name(s)
2 last = last_name(s)
3 return last + ',' + first
```

Call: last_name_first('Walker White'):



def first_name(s):

"""**Prec:** see last_name_first"""

```
1 end = s.find(' ')
2 return s[0:end]
```

Frames and Helper Functions

```
def last_name_first(s):
```

```
    """Precondition: s in the form  
    <first-name> <last-name>"""
```

```
1  first = first_name(s)  
2  last = last_name(s)  
3  return last + ',' + first
```

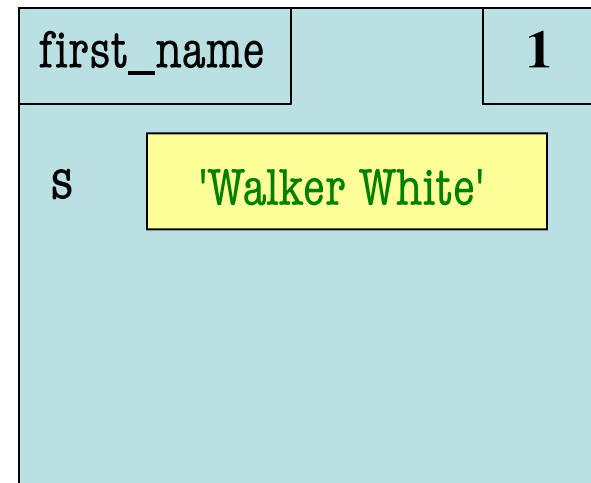
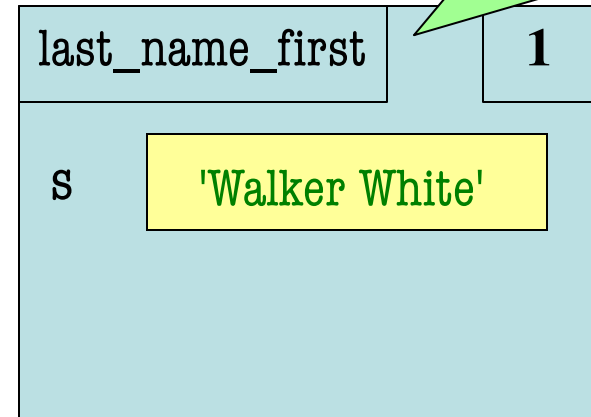
```
def first_name(s):
```

```
    """Prec: see last_name_first"""
```

```
1  end = s.find(' ')  
2  return s[0:end]
```

Call: last_name_first('Walker White')

Not done. Do not erase!



Frames and Helper Functions

def last_name_first(s):

"""**Precondition:** s in the form
<first-name> <last-name>"""

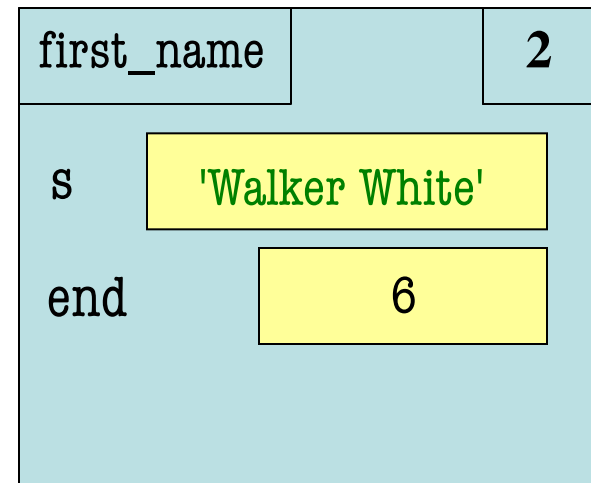
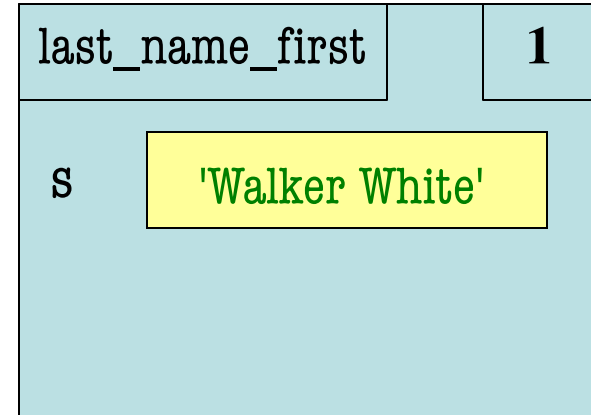
```
1 first = first_name(s)
2 last = last_name(s)
3 return last + ',' + first
```

def first_name(s):

"""**Prec:** see last_name_first"""

```
1 end = s.find(' ')
2 return s[0:end]
```

Call: last_name_first('Walker White'):



Frames and Helper Functions

def last_name_first(s):

"""**Precondition:** s in the form
<first-name> <last-name>"""

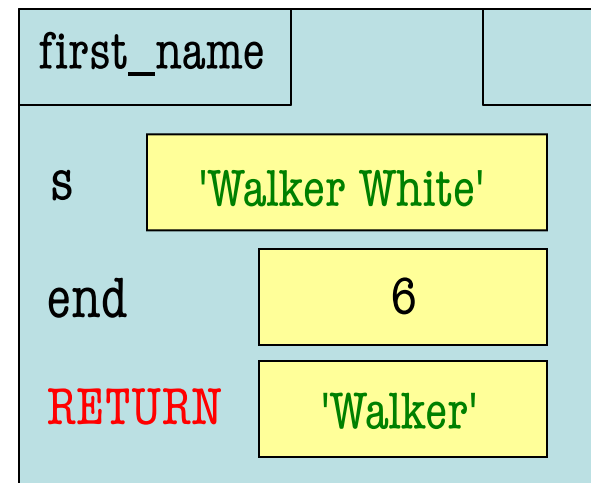
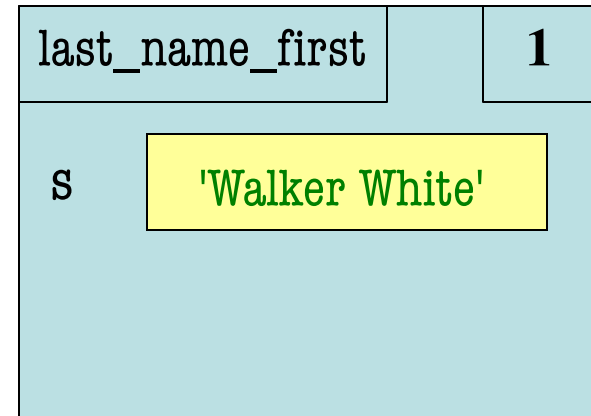
```
1 first = first_name(s)
2 last = last_name(s)
3 return last + ',' + first
```

def first_name(s):

"""**Prec:** see last_name_first"""

```
1 end = s.find(' ')
2 return s[0:end]
```

Call: last_name_first('Walker White'):



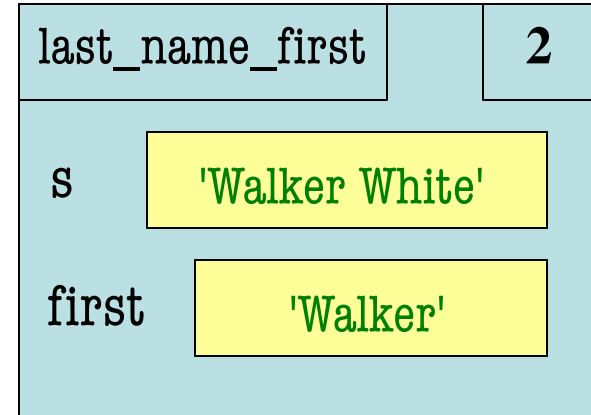
Frames and Helper Functions

def last_name_first(s):

"""**Precondition:** s in the form
<first-name> <last-name>"""

```
1 first = first_name(s)
2 last = last_name(s)
3 return last + ',' + first
```

Call: last_name_first('Walker White'):



def first_name(s):

"""**Prec:** see last_name_first"""

```
1 end = s.find(' ')
2 return s[0:end]
```

ERASE WHOLE FRAME

Frames and Helper Functions

def last_name_first(s):

"""**Precondition:** s in the form
<first-name> <last-name>"""

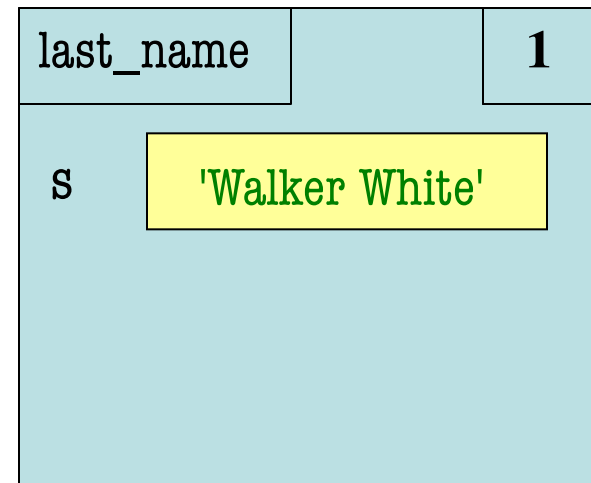
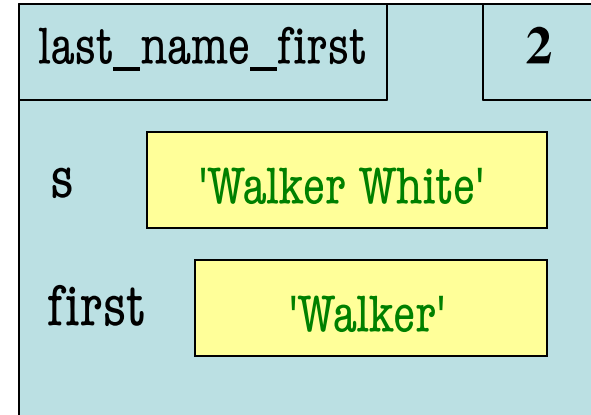
```
1 first = first_name(s)
2 last = last_name(s)
3 return last + '.' + first
```

def last_name(s):

"""**Prec:** see last_name_first"""

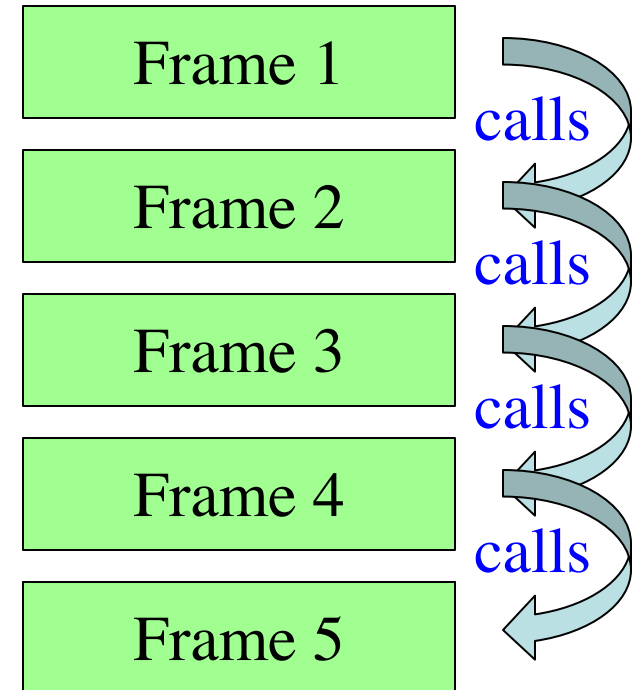
```
1 end = s.rfind(' ')
2 return s[end+1:]
```

Call: last_name_first('Walker White'):



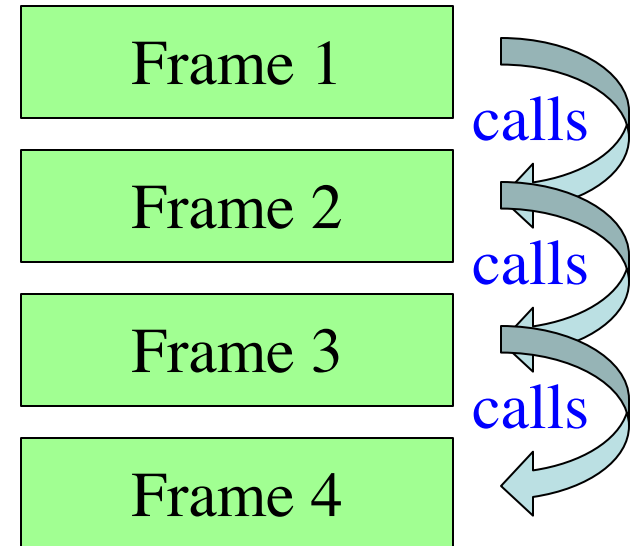
The Call Stack

- 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



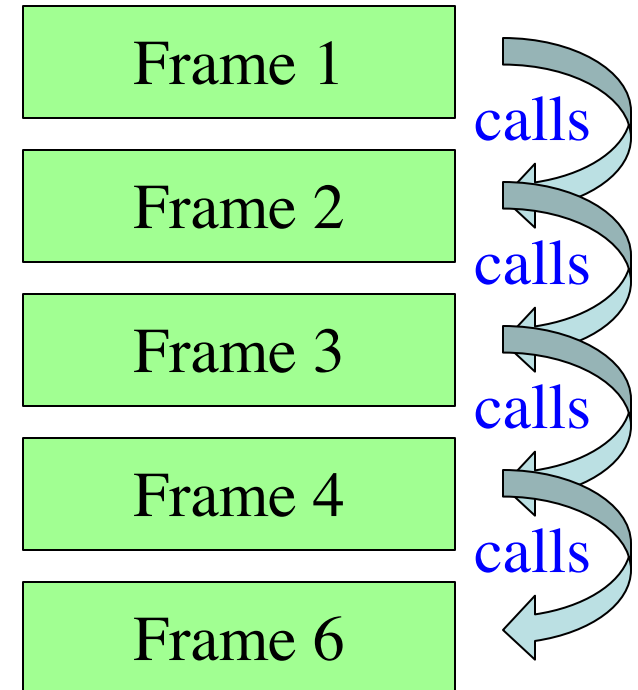
The Call Stack

- 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



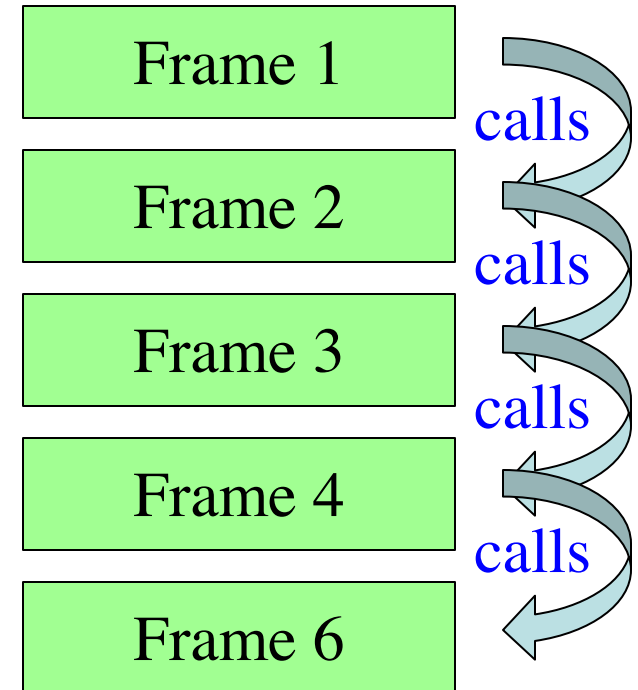
The Call Stack

- 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



The Call Stack

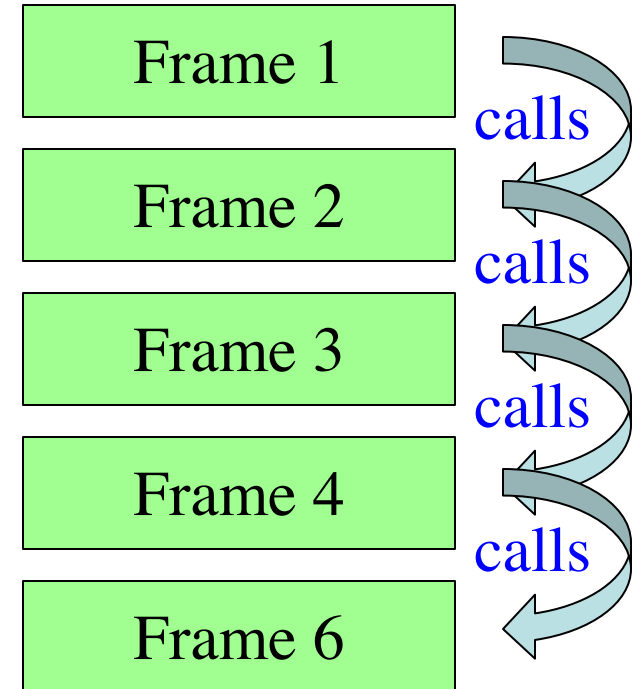
- Functions are “stacked”
 - Cannot remove one above w/o removing one below
 - Sometimes draw bottom up (better fits the metaphor)
- Stack represents memory as a “high water mark”
 - Must have enough to keep the **entire stack** in memory
 - Error if cannot hold stack



The Call Stack

- Functions are “stacked”
 - Can be called w/o a “frame” called module.
 - Some functions (built-in) are in the global space
- Stack represents memory as a “high water mark”
 - Must have enough to keep the **entire stack** in memory
 - Error if cannot hold stack

Book adds a special “frame” called module.
This is **WRONG!**
Module is global space



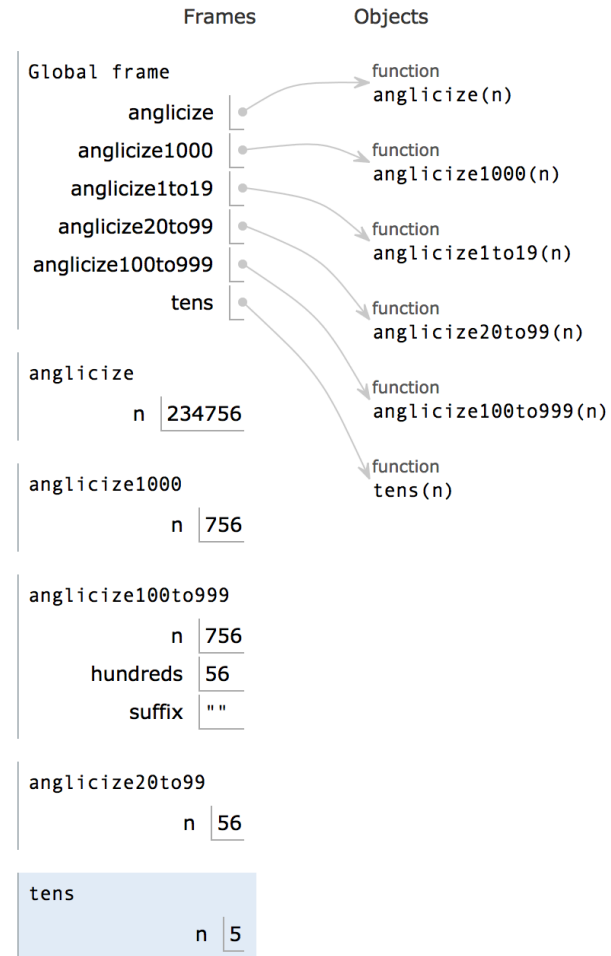
Anglicize Example

```
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

