Lecture 10

# Memory in Python

# Announcements For This Lecture

## Reading

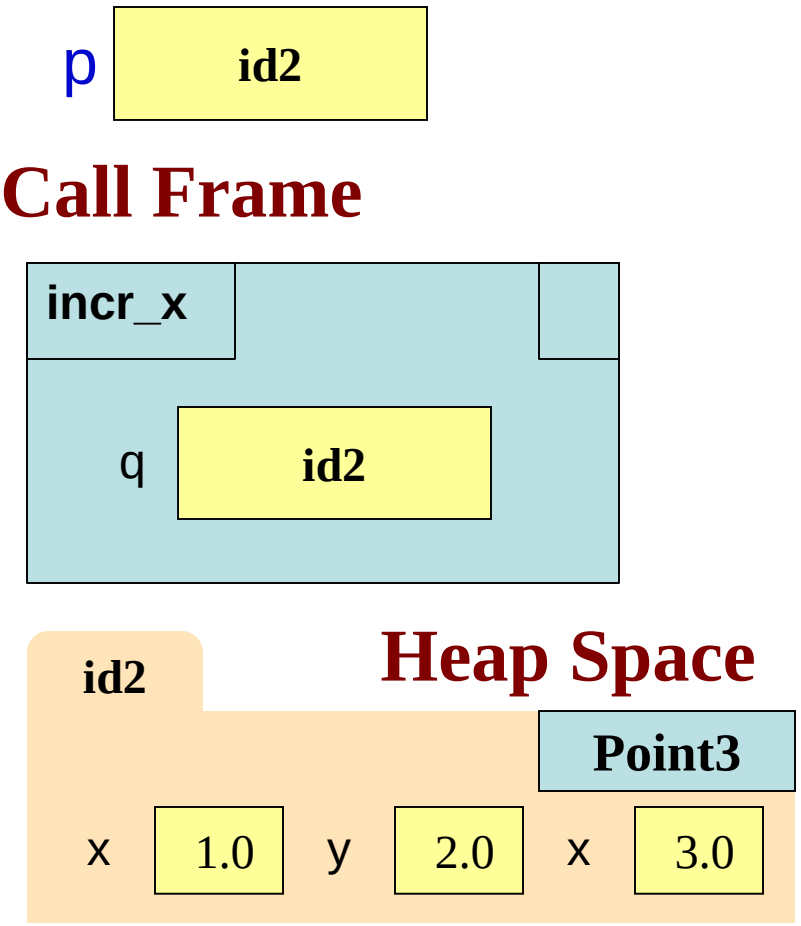
- Reread all of Chapter 3

## Assignments

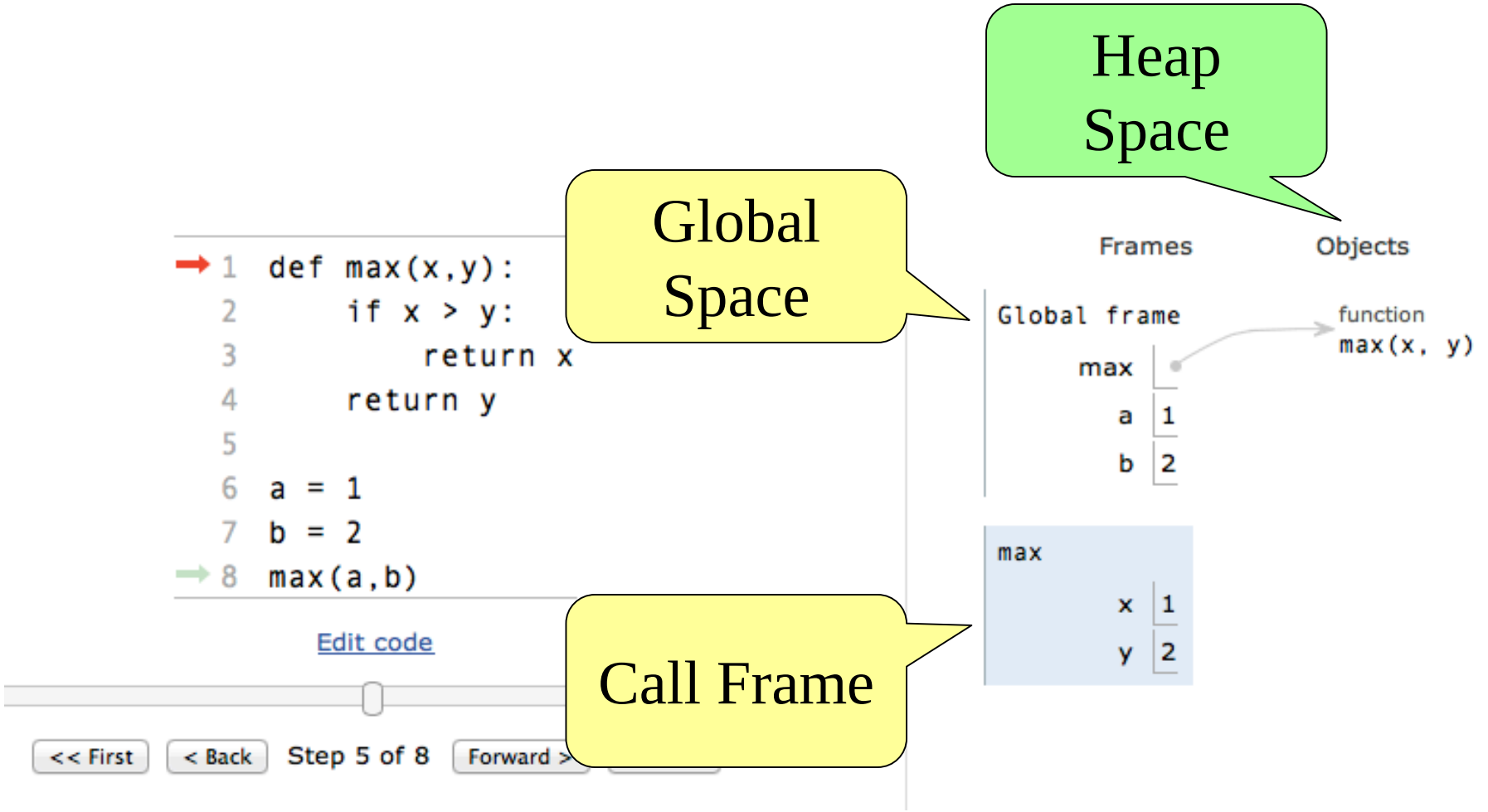
- Finishing Assignment 1
- Assignment 2 now online
  - Has objects, lists, dicts
  - Also if and for-loops
  - Due Sunday, March 12

# 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



# Memory and the Python Tutor



# Functions and Global Space

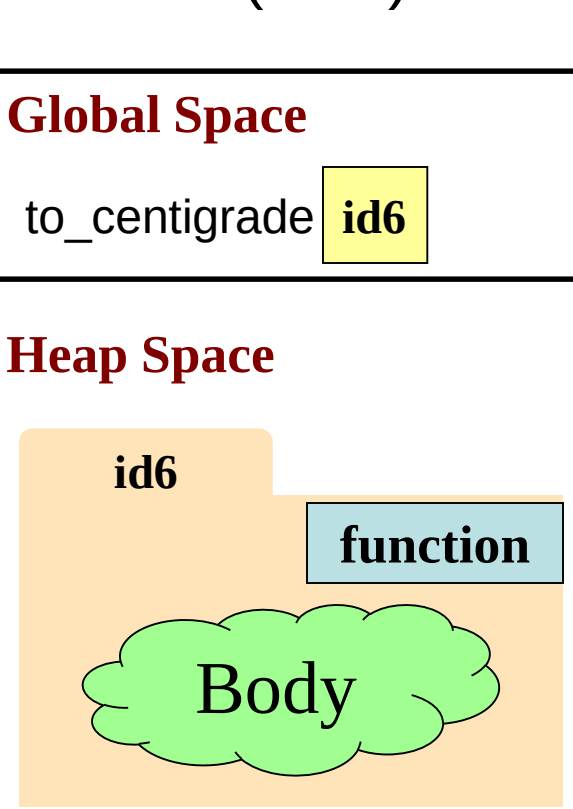
- A function definition…
  - Creates a global variable (same name as function)
  - Creates a **folder** for body
  - Puts folder id in variable
- Variable vs. Call

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

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

Body

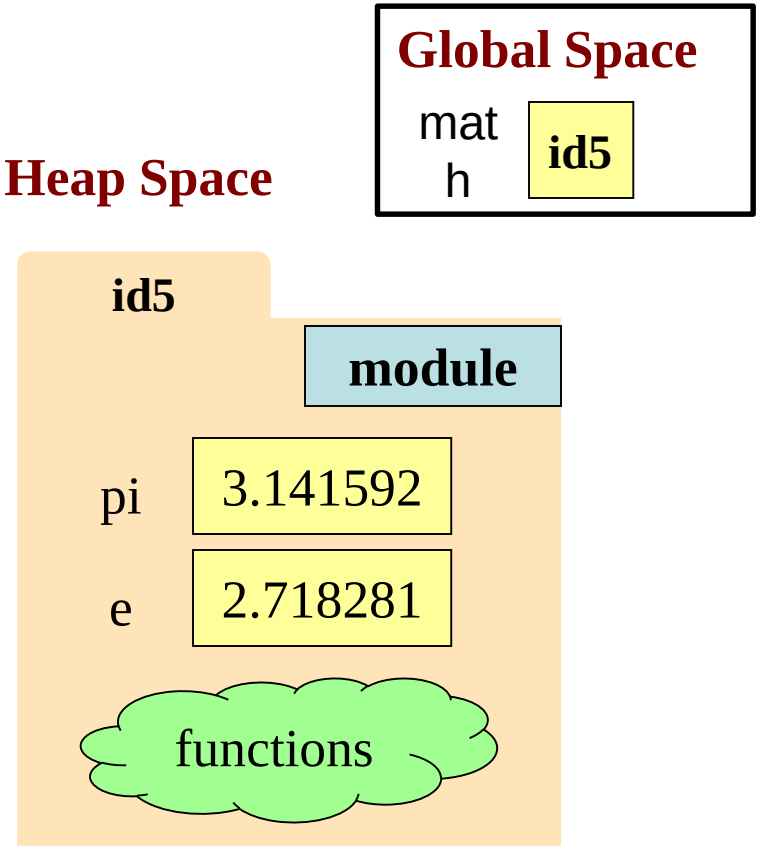


# Modules and Global Space

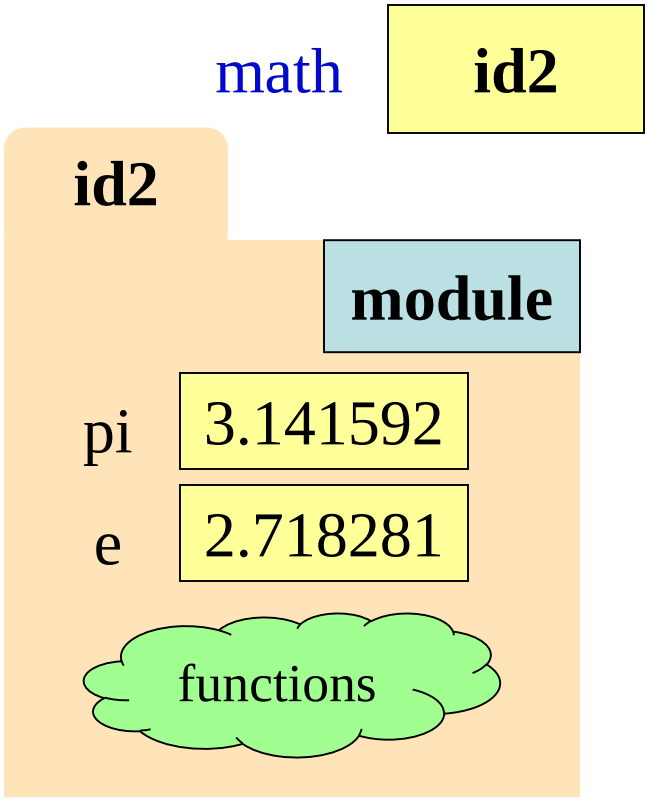
- Importing a module:
  - Creates a global variable (same name as module)
  - Puts contents in a **folder**
    - Module variables
    - Module functions
  - Puts folder id in variable
- **from** keyword dumps contents to global space

```
import math
```

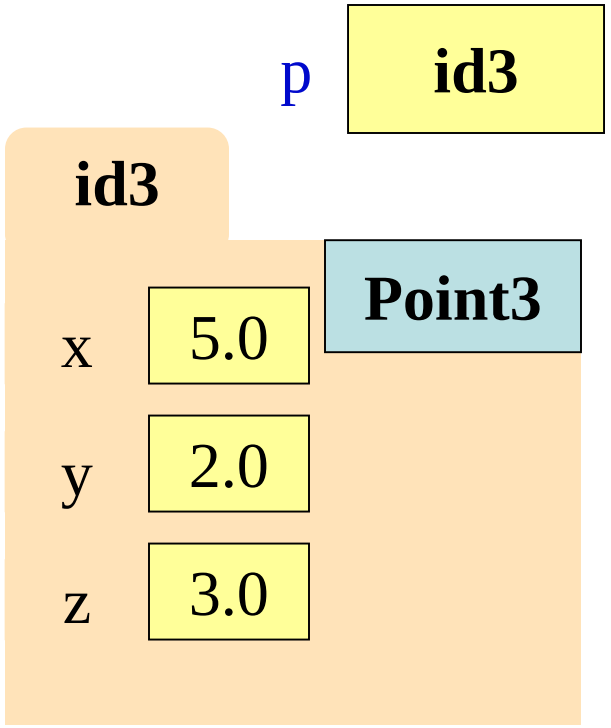


# Modules vs Objects

## Module

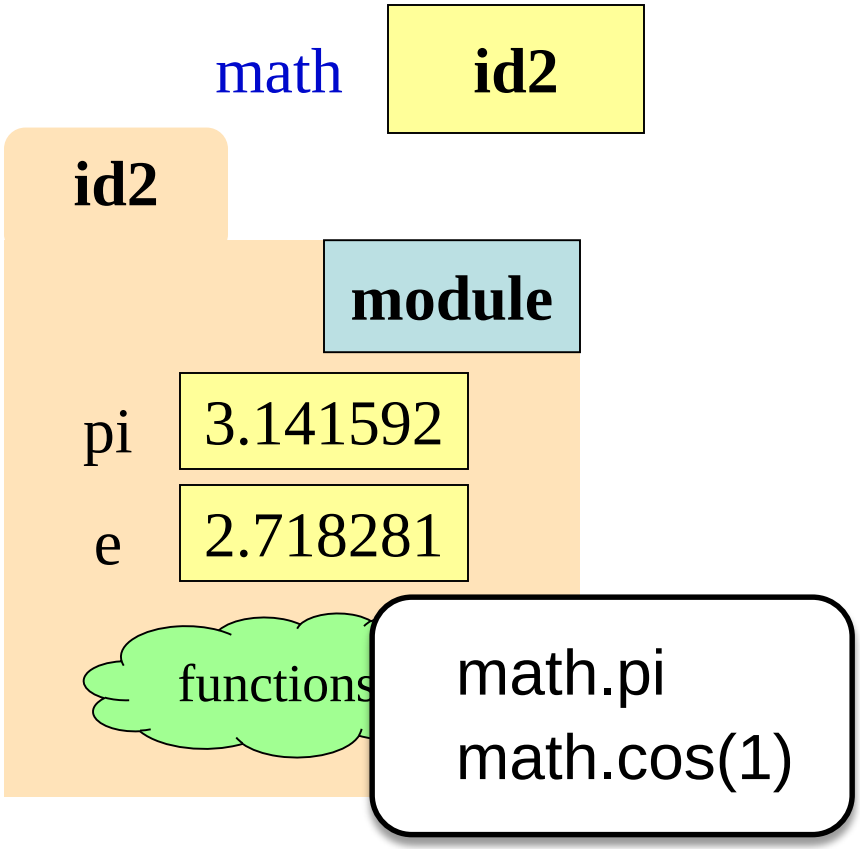



## Object

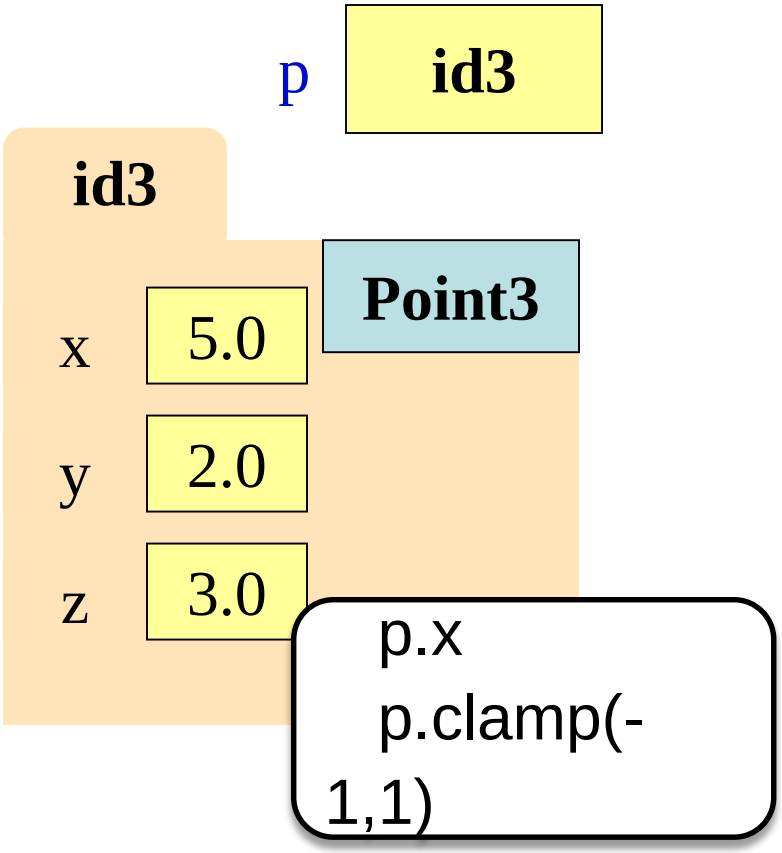


# Modules vs Objects

## Module




## Object



# Modules vs Objects

## Module

math | id2

id2

m

pi | 3.14159

e | 2.718281

functions

math.pi
math.cos(1)

## Object

p | id3

int3

y | 2.0

z | 3.0

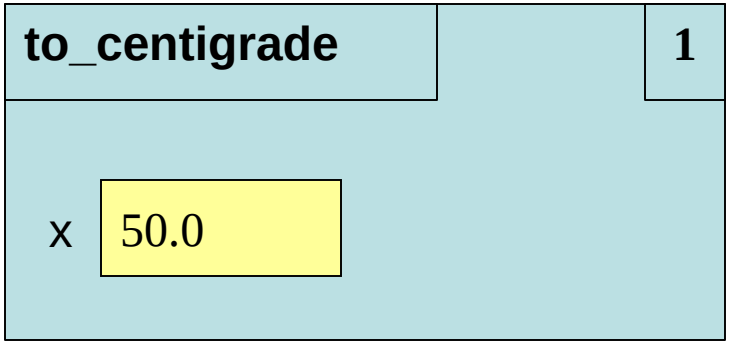
p.x
p.clamp(-1,1)

The period (.) means
"go inside of the folder"

# 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!

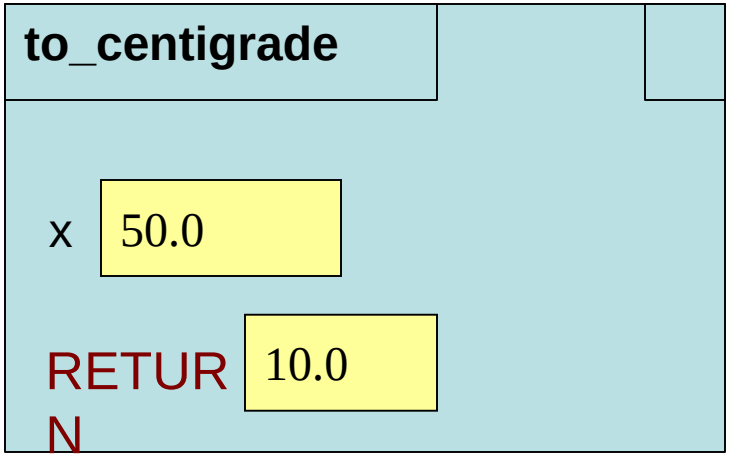
```
def to_centigrade(x):
    return 5*(x-32)/9.0
```

# 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

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

**Call**: to_centigrade(50.0)



# 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

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

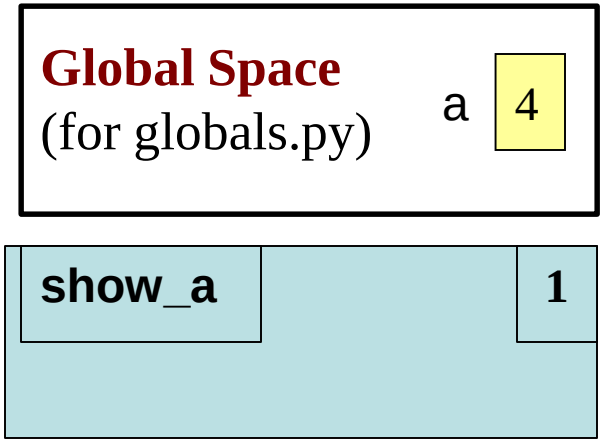
But don't actually erase on an exam

# 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

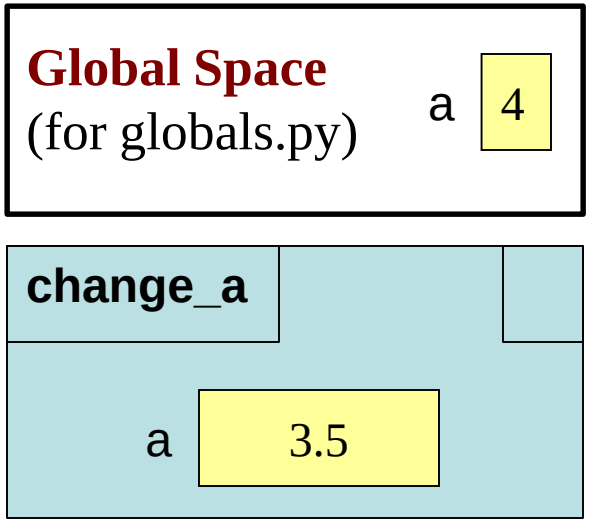
- All function definitions are in some module
- Call can access global space for **that module**
  - math.cos: global for math
  - temperature.to_centigrade uses global for temperature
- But **cannot** change values
  - Assignment to a global makes a new local variable!
  - Why we limit to constants




```
# 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



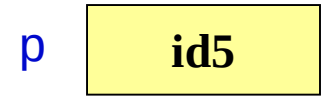

```
# globals.py
"""Show how globals work"""
a = 4 # global space

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

# Call Frames and Objects

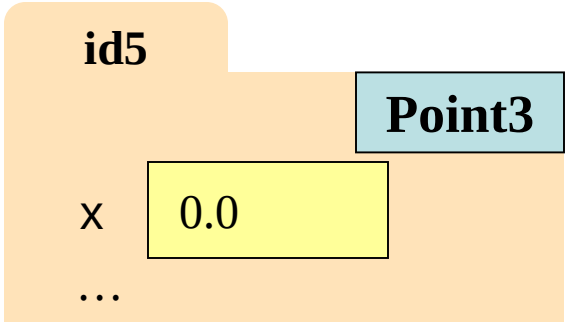
- Mutable objects can be altered in a function call
  - Object vars hold names!
  - Folder accessed by both global var & parameter
- **Example**:

```
def incr_x(q):
    q.x = q.x + 1
>>> p = Point3(0,0,0)
>>> incr_x(p)
```

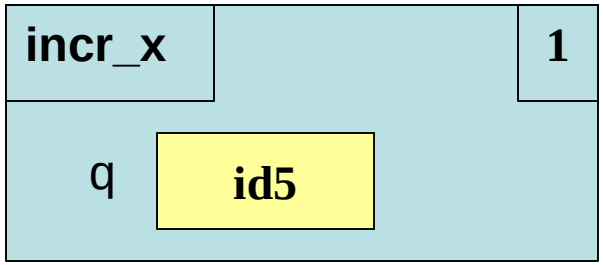
**Global Space**




**Heap Space**




**Call Frame**

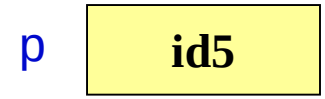


# Call Frames and Objects

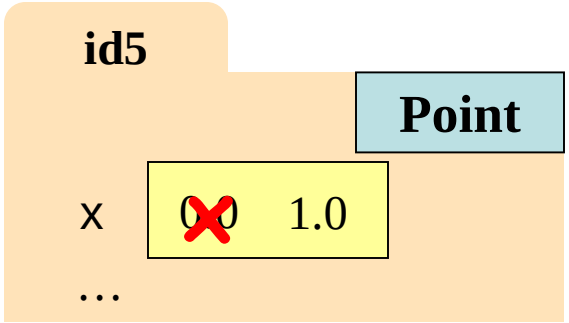
- Mutable objects can be altered in a function call
  - Object vars hold names!
  - Folder accessed by both global var & parameter
- **Example**:

```
def incr_x(q):
    q.x = q.x + 1
>>> p = Point(0,0,0)
>>> incr_x(p)
```

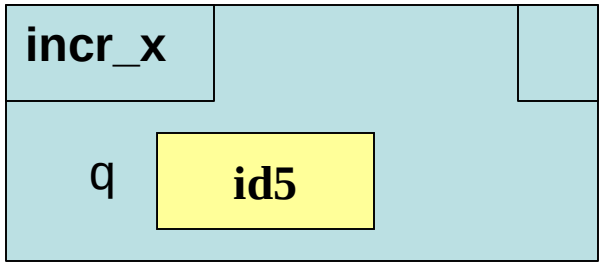
**Global Space**




**Heap Space**




**Call Frame**

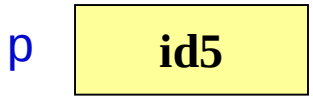


# Call Frames and Objects

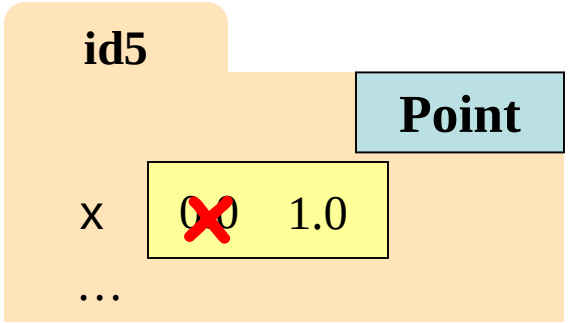
- Mutable objects can be altered in a function call
  - Object vars hold names!
  - Folder accessed by both global var & parameter
- **Example**:

```
def incr_x(q):
    q.x = q.x + 1
>>> p = Point(0,0,0)
>>> incr_x(p)
```

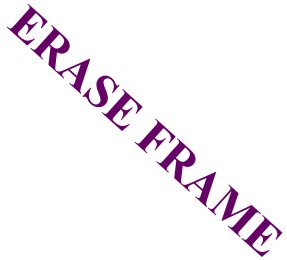
**Global Space**




**Heap Space**



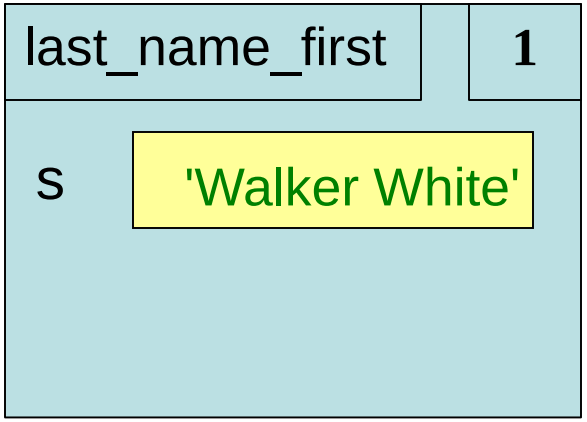

**Call Frame**



# Frames and Helper Functions

```
def last_name_first(s):
    """Precondition: s in the form
    <first-name> <last-name>"""
    first = first_name(s)
    last = last_name(s)
    return last + ',' + first
```

**Call:** last_name_first('Walker White'):




```
def first_name(s):
    """Prec: see last_name_first"""
    end = s.find(' ')
    return s[0:end]
```

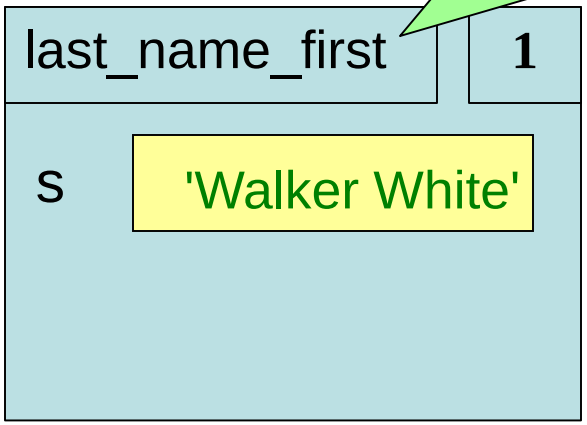
# Frames and Helper Functions

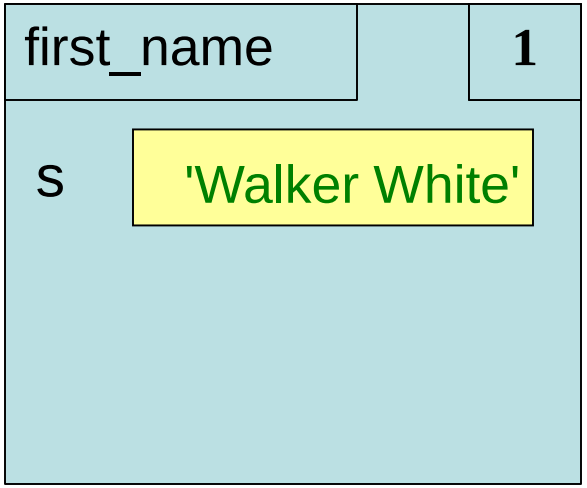
```
def last_name_first(s):
    """Precondition: s in the form
    <first-name> <last-name>"""
1   first = first_name(s)
2   last = last_name(s)
3   return last + ',' + first
```

```
def first_name(s):
    """Prec: see last_name_first"""
1   end = s.find(' ')
2   return s[0:end]
```

**Call:** last_name_first(...)

Not done. Do not erase!



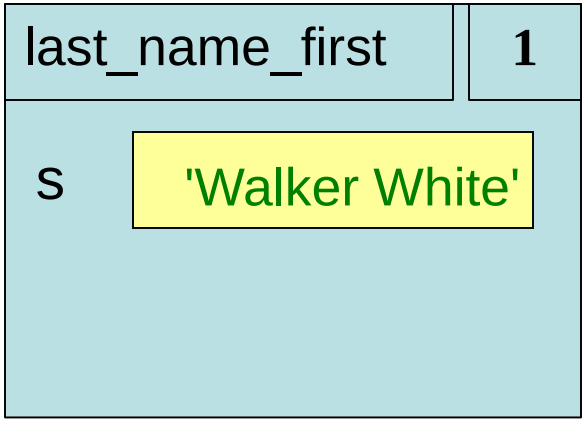



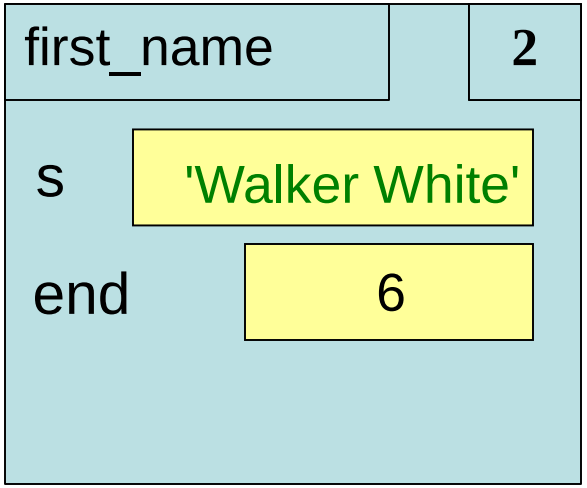
# Frames and Helper Functions

```
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'):



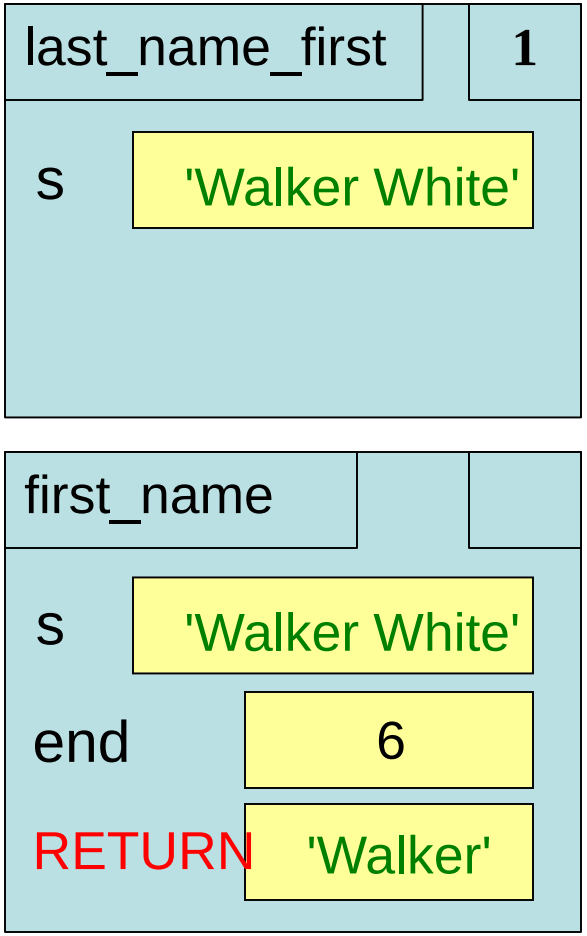



# Frames and Helper Functions

```
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]
```

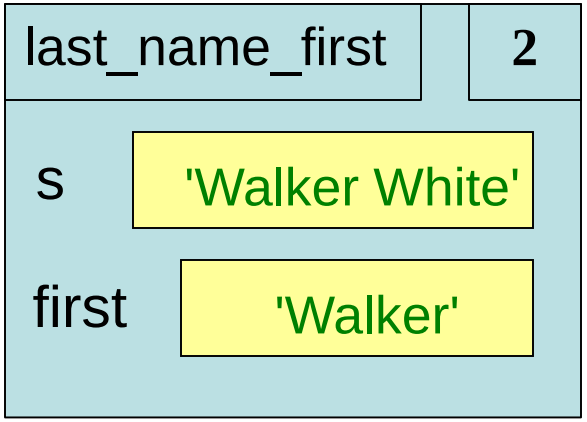
**Call:** last_name_first('Walker White'):



# 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]
```

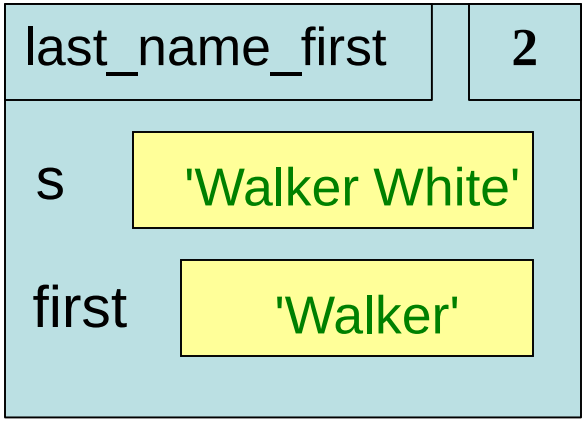
ERASE WHOLE FRAME

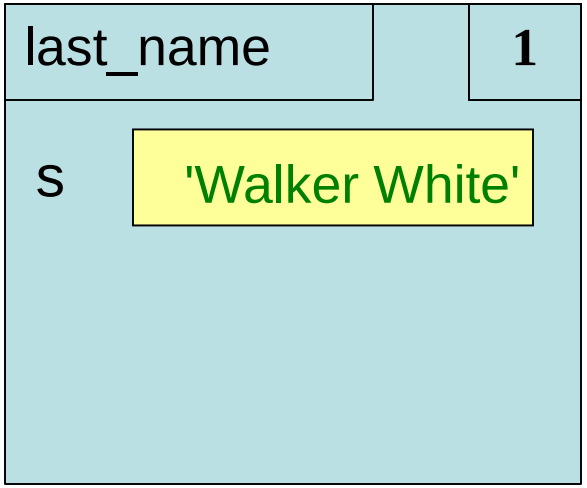
# Frames and Helper Functions

```
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:]
```

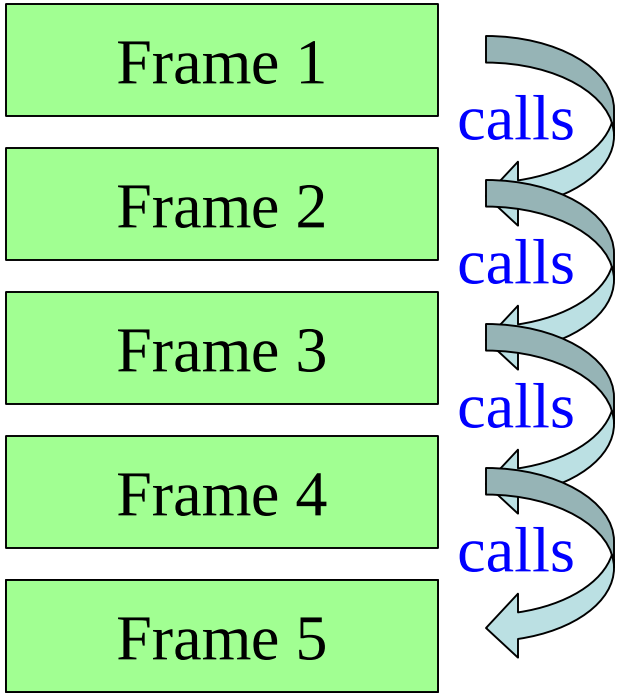
**Call:** last_name_first('Walker White'):



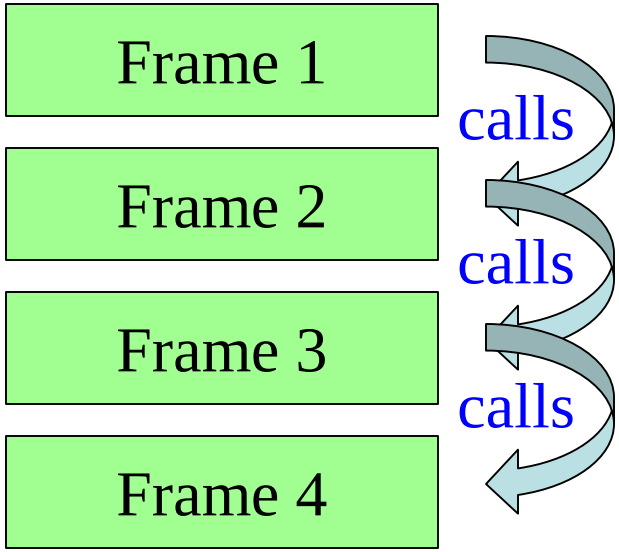
# The Call Stack

- 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



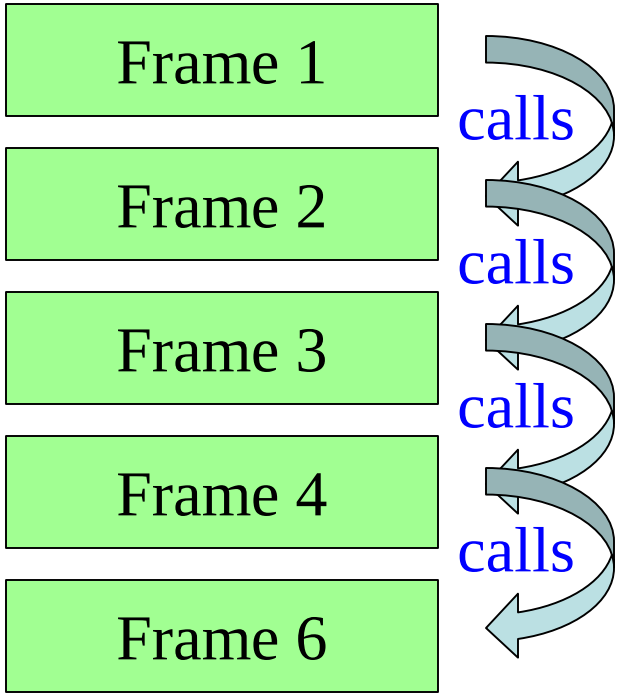
# The Call Stack

- 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



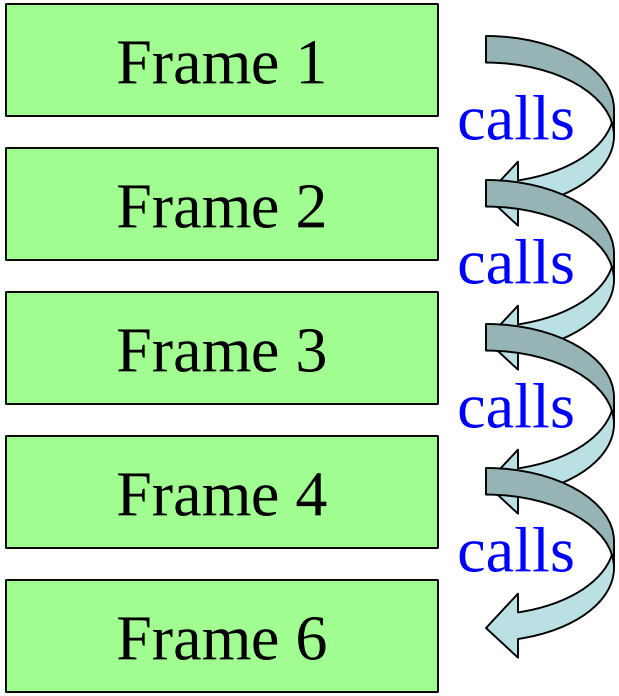
# The Call Stack

- 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

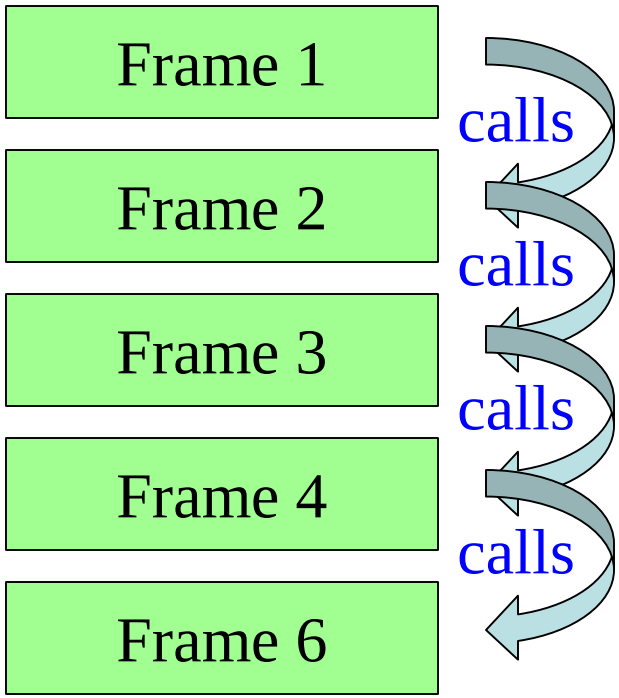
- Functions are "stacked"
  - Cannot remove one above w/o removing one below
  - Sometimes draw bottom up (better fits the metaphor)
- Stack represents memory as a "high water mark"
  - Must have enough to keep the **entire stack** in memory
  - Error if cannot hold stack



# The Call Stack

- Functions are "stacked"
  - Ca
    w/
  - So
    (b
- 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

```
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'
```

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

line that has just executed
next line to execute

Frames

Global frame
- anglicize
- anglicize1000
- anglicize1to19
- anglicize20to99
- anglicize100to999
- tens

anglicize
- n: 234756

anglicize1000
- n: 756

anglicize100to999
- n: 756
- hundreds: 56
- suffix: ""

anglicize20to99
- n: 56

tens
- n: 5

Objects

- function anglicize(n)
- function anglicize1000(n)
- function anglicize1to19(n)
- function anglicize20to99(n)
- function anglicize100to999(n)
- function tens(n)

# Anglicize Example

```
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'
```

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

line that has just executed
next line to execute

Frames

Global frame
- anglicize
- anglicize1000
- anglicize1to19 → function anglicize1to19(n)
- anglicize20to99 → function anglicize20to99(n)
- anglicize100to999
- tens

Global Space

anglicize
- n: 234756

anglicize1000
- n: 756

anglicize100to999
- n: 756
- hundreds: 56
- suffix: ""

anglicize20to99
- n: 56

tens
- n: 5

Call Stack