Lecture 9

Call Stacks

Announcements for Today

Reading

- Reread Chapter 3
- 10.0-10.2, 10.4-10.6 for Tue

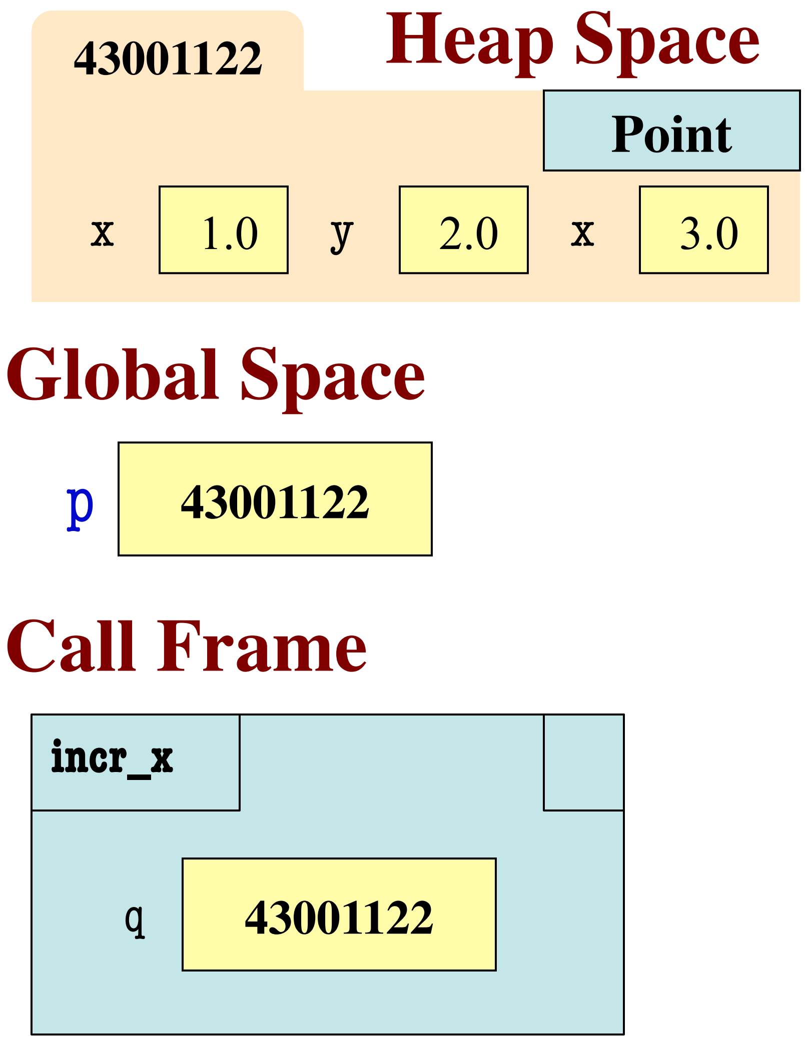
- **Prelim, Oct 4th 7:30-9:30**
  - Material up to next Tuesday
  - Study guide next week
- **Conflict with Prelim time?**
  - Submit to Prelim 1 Conflict assignment on CMS
  - Do not submit if no conflict

Assignments

- Work on your revisions
  - Want done by Monday
- Assignment 2 Tuesday
  - Due IN CLASS
  - Get help now if need it
- Assignment 3 posted
  - Due in two stages
  - Part 1 due Oct. 1 (pass/fail)
  - Part 2 due Oct. 11 (graded)

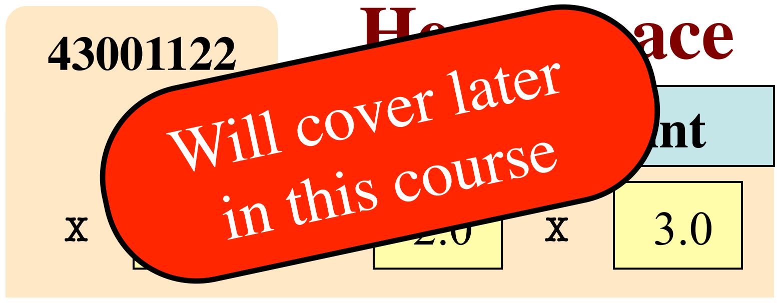
Modeling Storage in Python

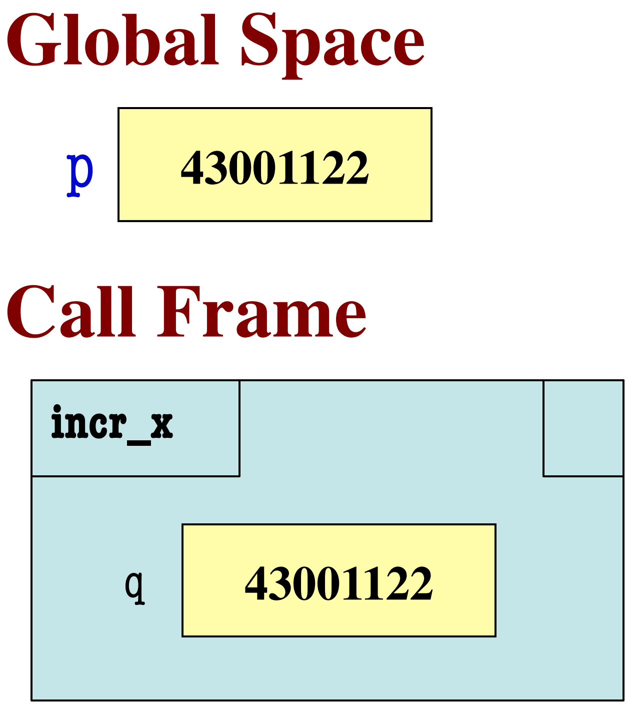
- **Call Frame**
  - Variables in function call
  - Deleted when call done
- **Global Space**
  - Global variables
  - Also **function names!**
  - All last until you quit
- **Heap Space**
  - Where "folders" are stored
  - Have to access indirectly



Modeling Storage in Python

- **Call Frame**
  - Variables in function call
  - Deleted when call done
- **Global Space**
  - Global variables
  - Also **function names!**
  - All last until you quit
- **Heap Space**
  - Where "folders" are stored
  - Have to access indirectly



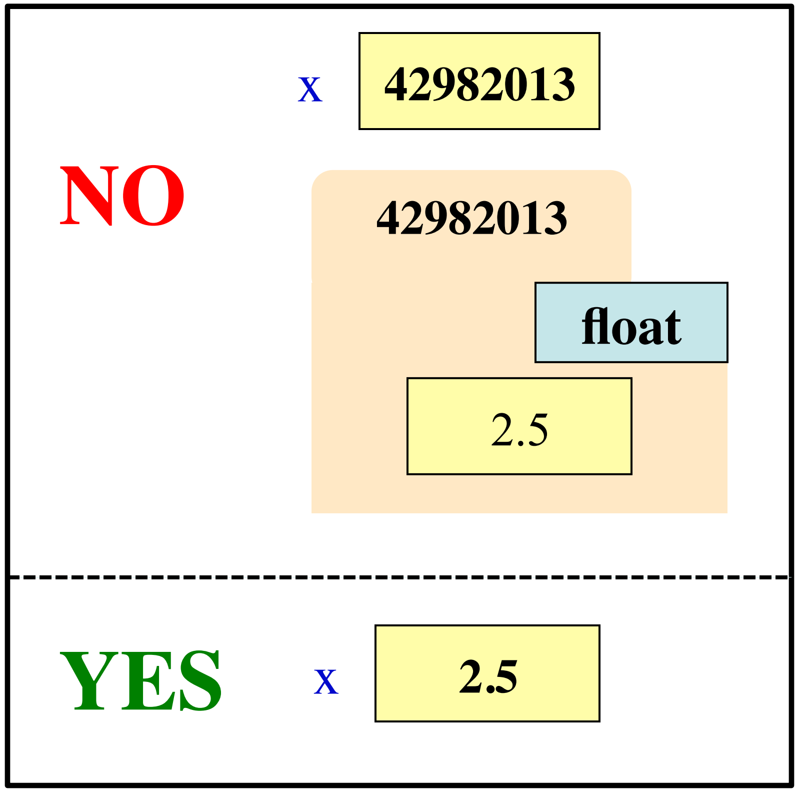


When Do We Need to Draw a Folder?

- When the value **contains** other values
  - This is what we are calling 'objects'
- When the value is **mutable**

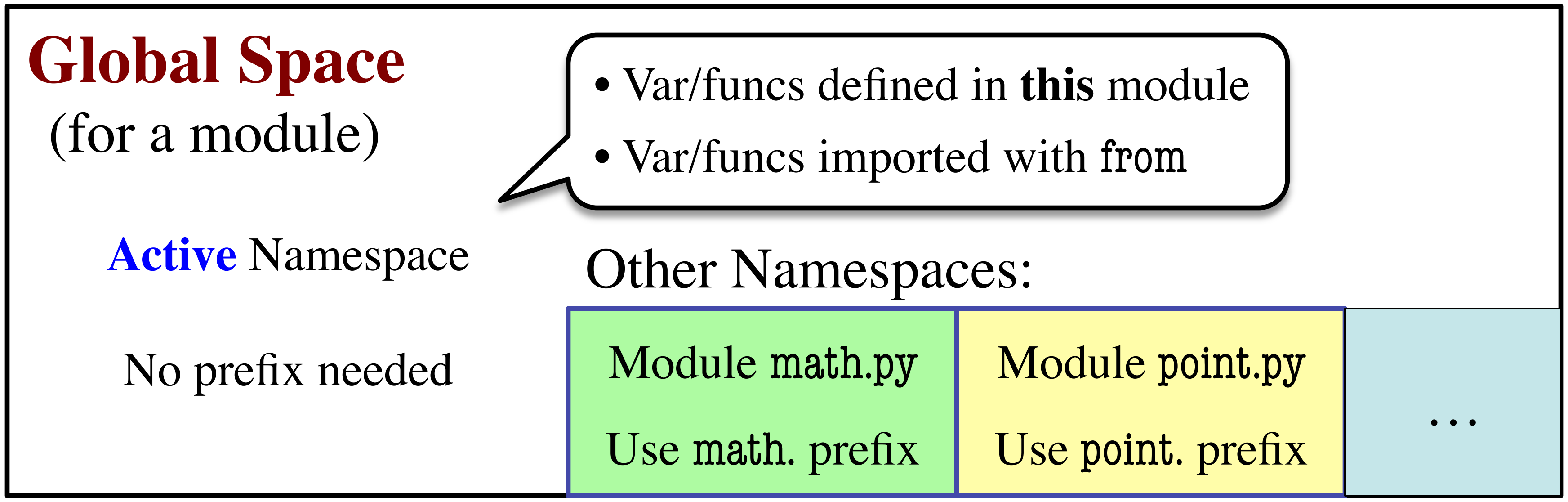
Type	Container?	Mutable?
int	No	No
float	No	No
str	Yes*	No
Point	Yes	Yes
RGB	Yes	Yes

* Contains characters, which is not a stand-alone type



Structure of Global Space

- Global space is defined relative to a **module**
  - Module you run with command `python <filename>`
  - Interactive prompt >>> is also a module with no name
- Global space is broken up into *namespaces*
  - Variables and functions for each imported module

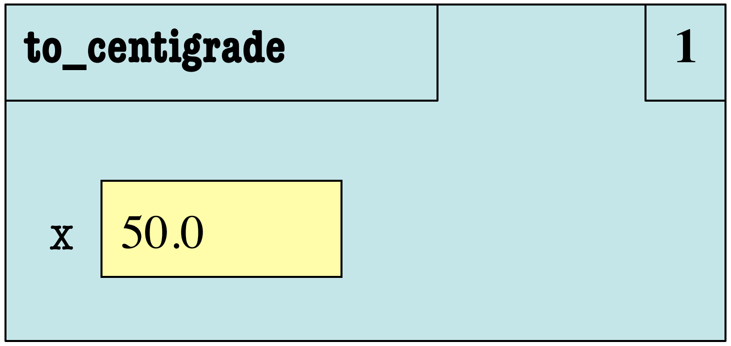


Review: Call Frames

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

Call: to_centigrade(50.0)

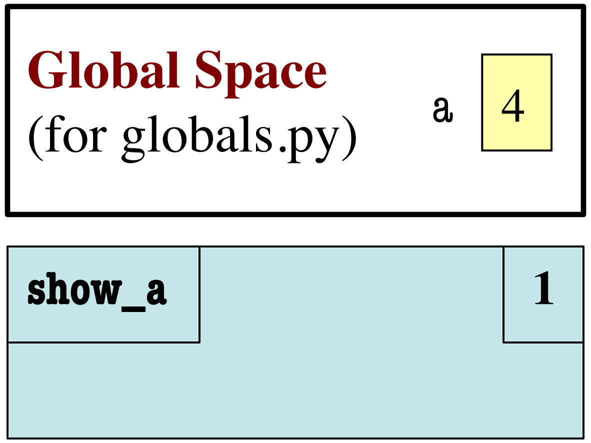


What is happening here?

Only at the End!

Function Access to Global Space

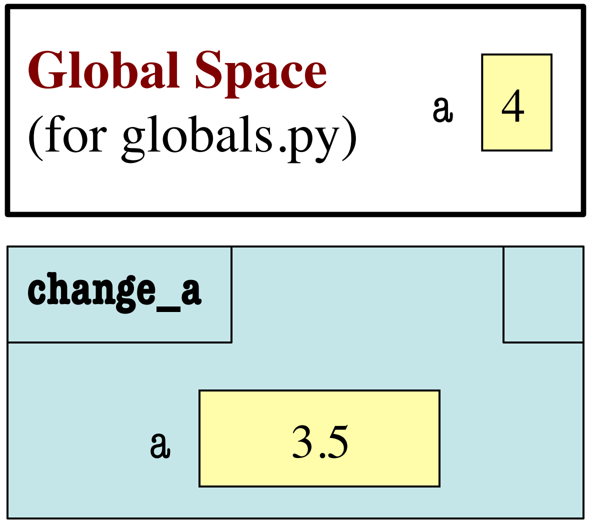
- All function definitions are in some module
- Call can access global space for **that module**
    - math.cos: global for math
    - temperature.to_centigrade uses global for temperature
- But **cannot** change values
    - Assignment to a global makes a new local variable!
    - Why we limit to constants



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


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

Function Access to Global Space

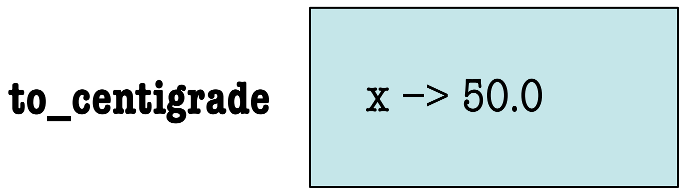
- All function definitions are in some module
- Call can access global space for **that module**
    - math.cos: global for math
    - temperature.to_centigrade uses global for temperature
- But **cannot** change values
    - Assignment to a global makes a new local variable!
    - Why we limit to constants



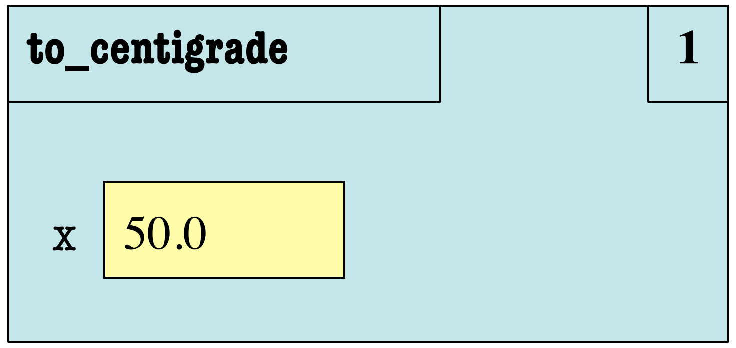
```
# globals.py
"""Show how globals work"""
a = 4 # global space


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

Text (Section 3.10) vs. Class

Textbook



This Class



Definition:

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

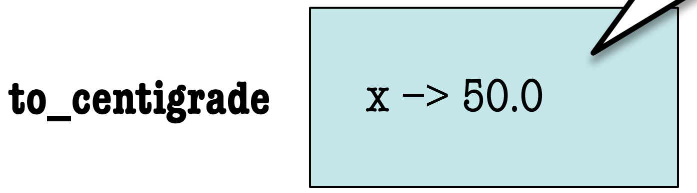
Call: to_centigrade(50.0)

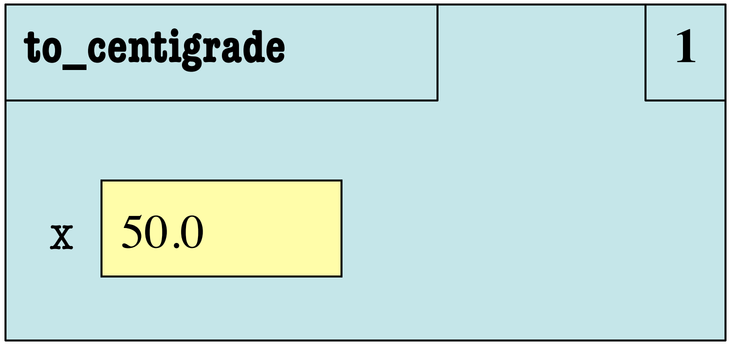
Text (Section 3.10) vs. Class

No instruction counter

Variables are not boxes

Textbook | **s Class**





Definition:

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

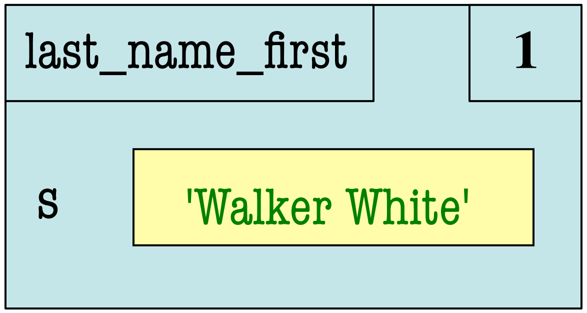
Call: to_centigrade(50.0)

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

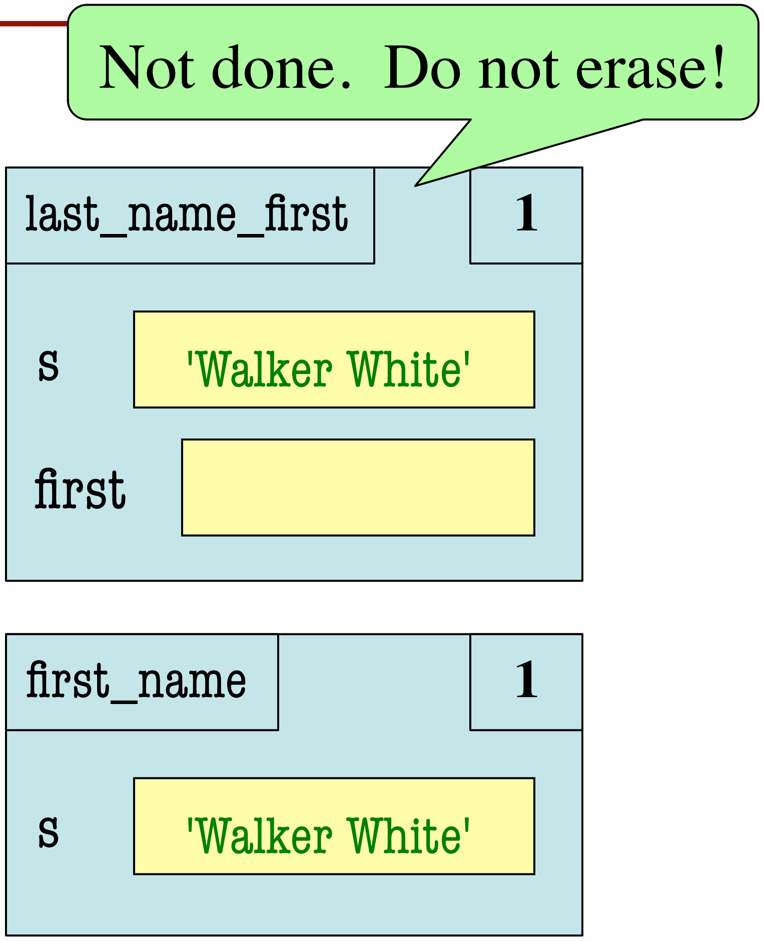
Frames and Helper Functions

```
   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]
```



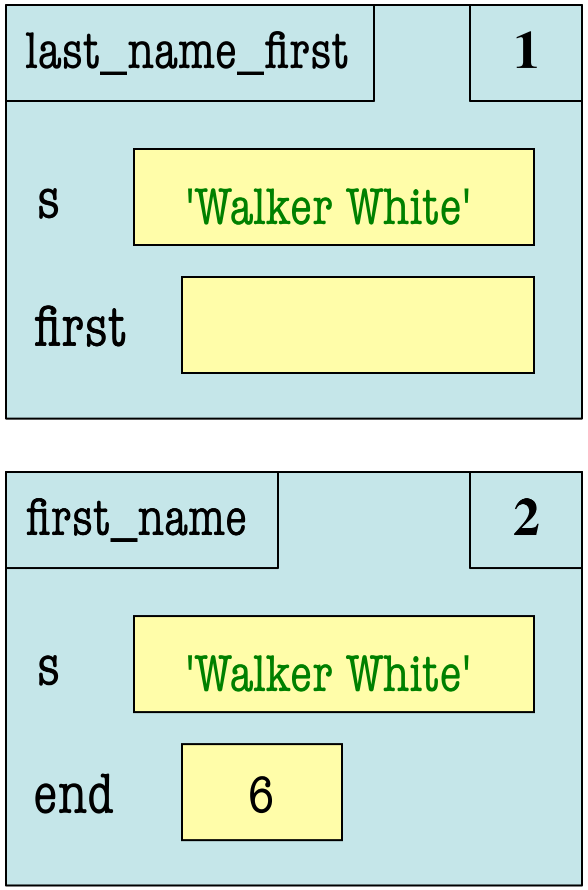
Frames and Helper Functions

```
   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]
```



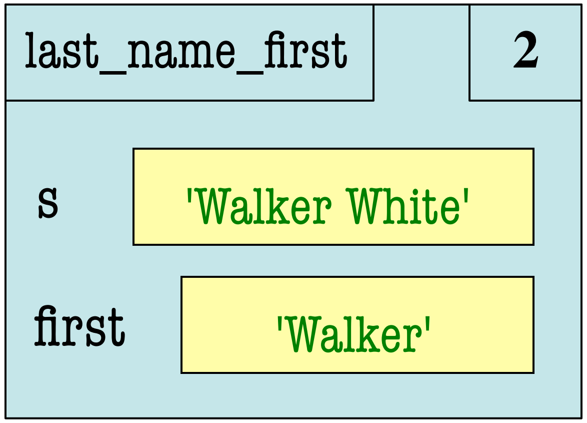
Frames and Helper Functions

```
  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]
```



Frames and Helper Functions

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

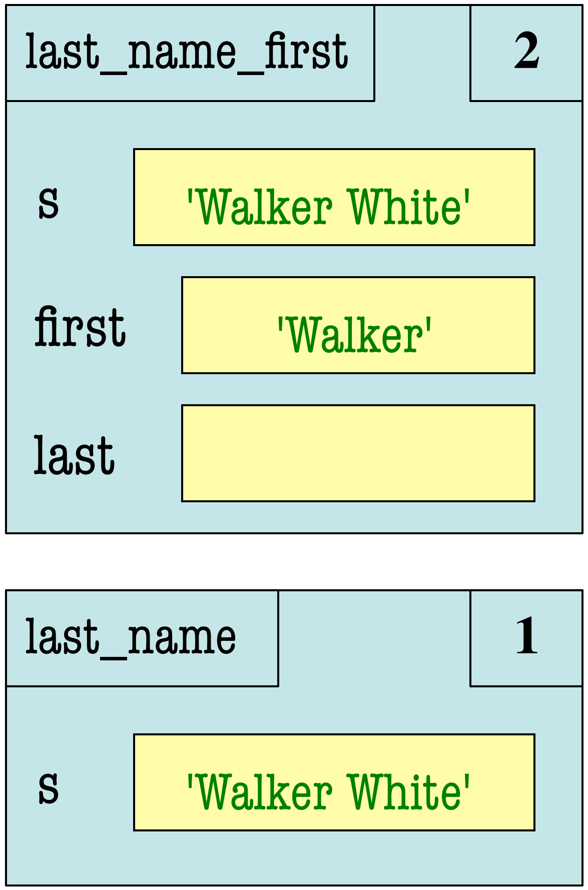


```
def  first_name(s):
        """Prec: see last_name_first"""
1       end = s.find(' ')
2       return s[0:end]
```

ERASE WHOLE FRAME

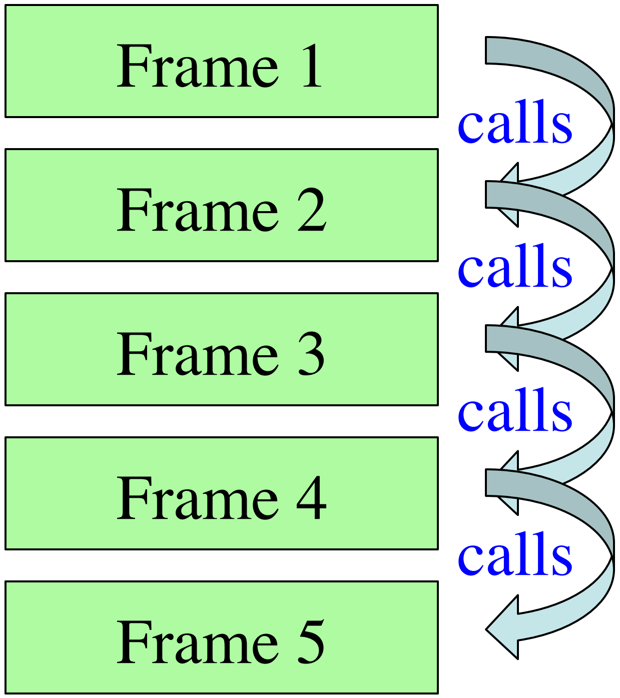
Frames and Helper Functions

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



The Call Stack

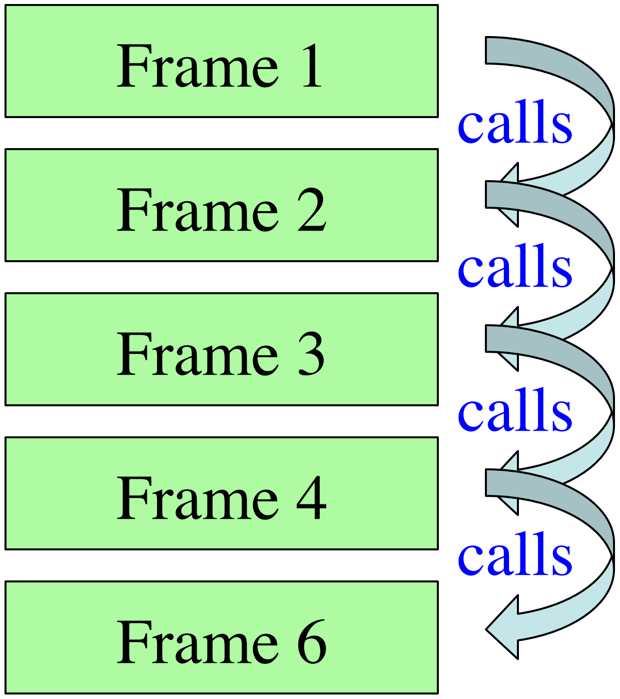
- Functions are "stacked"
  - Cannot remove one above w/o removing one below
  - Sometimes draw bottom up (better fits the metaphor)
- Stack represents memory as a "high water mark"
  - Must have enough to keep the **entire stack** in memory
  - Error if cannot hold stack



The Call Stack

- Functions are "stacked"
  - Can
    w/o
  - Son
    (bet
- 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



Errors and the Call Stack

```
# error.py

def function_1(x,y):
    return function_2(x,y)

def function_2(x,y):
    return function_3(x,y)

def function_3(x,y):
    return x/y # crash here

if __name__ == '__main__':
    print function_1(1,0)
```

When you crash, get the call stack:

```
Traceback (most recent call last):
  File "error.py", line 20, in <module>
    print function_1(1,0)
  File "error.py", line 8, in function_1
    return function_2(x,y)
  File "error.py", line 12, in function_2
    return function_3(x,y)
  File "error.py", line 16, in function_3
    return x/y
```

Make sure you can see line numbers in Komodo. Preferences ➔ Editor

Errors and the Call Stack

```
#

d
    return function_2(x,y)

def function_2(x,y):
    return function_3(x,y)

def function_3(x,y):
    return x/y # crash here

if
```

Application code. Not a frame!

Where error occurred (or where was found)

When you crash, get the call stack:

```
Traceback (most recent call last):
  File "error.py", line 20, in <module>
    print function_1(1,0)
  File "error.py", line 8, in function_1
    return function_2(x,y)
  File "error.py", line 12, in function_2
    return function_3(x,y)
  File "error.py", line 16, in function_3
    return x/y
```

Make sure you can see line numbers in Komodo. Preferences ➔ Editor

Assert Statements

```
assert <boolean>              # Creates error if <boolean> false
assert <boolean>, <string>    # As above, but displays <String>
```

- Way to force an error
  - Why would you do this?
- Enforce preconditions!
  - Put precondition as assert.
  - If violate precondition, the program crashes
- Provided code in A3 uses asserts heavily

```
def exchange(amt, from_c, to_c)
    """Returns: amt from exchange
       Precondition: amt is a float..."""
    assert type(amt) == float
    …
```

See asserts.py for more

Recovering from Errors

- try-except blocks allow us to recover from errors
  - Do the code that is in the try-block
  - Once an error occurs, jump to the catch
- **Example**:

```
try:
    input = raw_input() # get number from user
    x = float(input)      # convert string to float
    print 'The next number is '+`x+1`
except:
    print 'Hey! That is not a number!'
```

might have an error

executes have an error

Recovering from Errors

- try-catch blocks allow us t
  - Do the code that is in the tr
  - Once an error occurs, jump
- **Example**:

```
try:
    input = raw_input() # get number from user
    x = float(input)    # convert string to float
    print 'The next number is '+`x+1`
except:
    print 'Hey! That is not a number!'
```

- Similar to if-else
  - Except always does try
  - Just might not do **all** of the try block

Try-Except is Very Versatile

```
def isfloat(s):
    """Returns: true if string s
    represents a float"""
    try:
        x = float(s)
        return True
    except:
        return False
```

Conversion to a float might fail

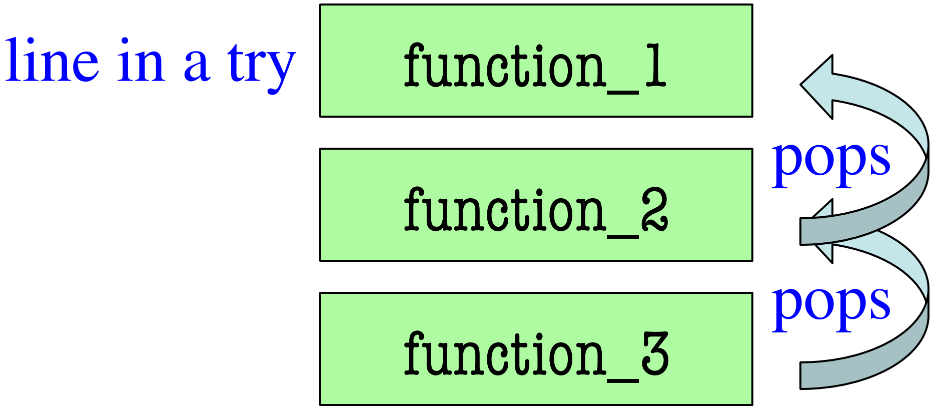
If attempt succeeds, string s is a float

Otherwise, it is not

Try-Except and the Call Stack

```
# recover.py

def function_1(x,y):
    try:
        return function_2(x,y)
    except:
        return float('inf')

def function_2(x,y):
    return function_3(x,y)

def function_3(x,y):
    return x/y # crash here
```

- Error "pops" frames off stack
  - Starts from the stack bottom
  - Continues until it sees that current line is in a try-block
  - Jumps to except, and then proceeds as if no error



Try-Except and the Call Stack

```
# recover.py

def function_1(x,y):
    try:
        return function_2(
    except:
        return float('inf')

def function_2(x,y):
    return function_3(x,y)

def function_3(x,y):
    return x/y # crash here
```

How to return
∞ as a float.

- Error "pops" frames off stack
  - om the stack bottom
  - es until it sees that current line is in a try-block
  - Jumps to except, and then proceeds as if no error
- **Example**:

```
>>> print function_1(1,0)
inf
>>>
```

No traceback!