CS 1110:
Introduction to Computing Using Python

Lecture 9

Memory in Python

[Andersen, Gries, Lee, Marschner, Van Loan, White]

Announcements: Assignment 1

- A1 is graded. If your A1 is not perfect, your first grade is a 1.
  - This is a counter for how many times you have submitted.
  - It is not a permanent grade, can resubmit.
- **In order to give students more chances to revise, the March 2nd resubmit deadline is being extended until Sunday, March 5th 11:59pm**
- Review the announcements from the end of Lecture 6 for policies:

http://www.cs.cornell.edu/courses/cs1110/2017sp/lectures/02-14-17/presentation-06.pdf

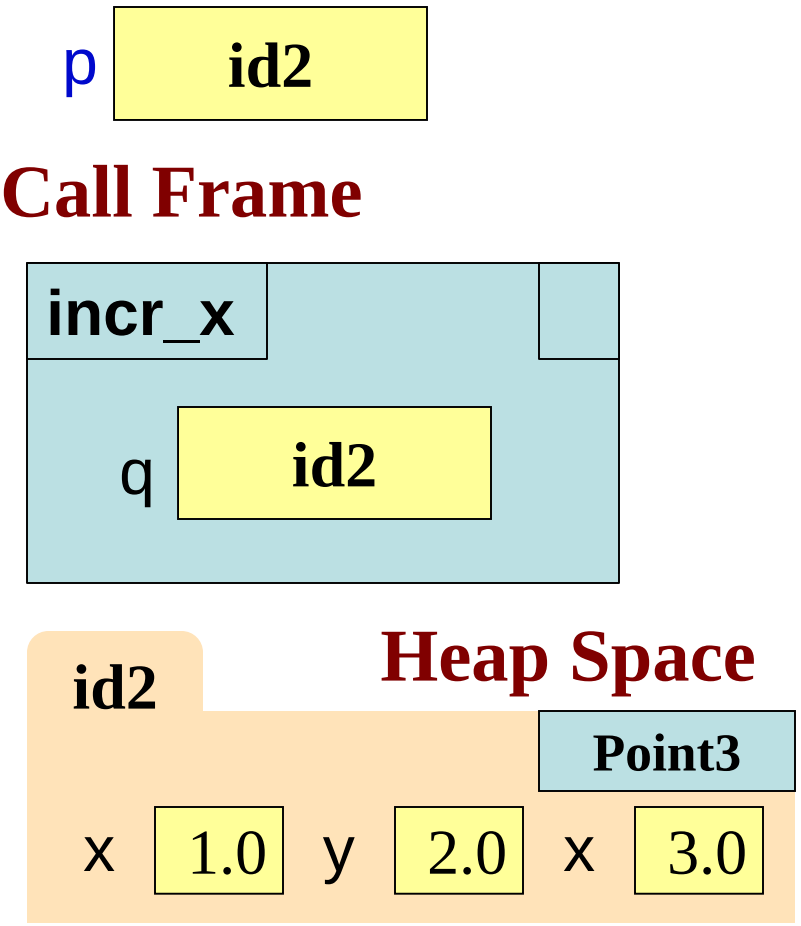
- Read Section 2.3 of A1 carefully to understand how to revise.

Announcements

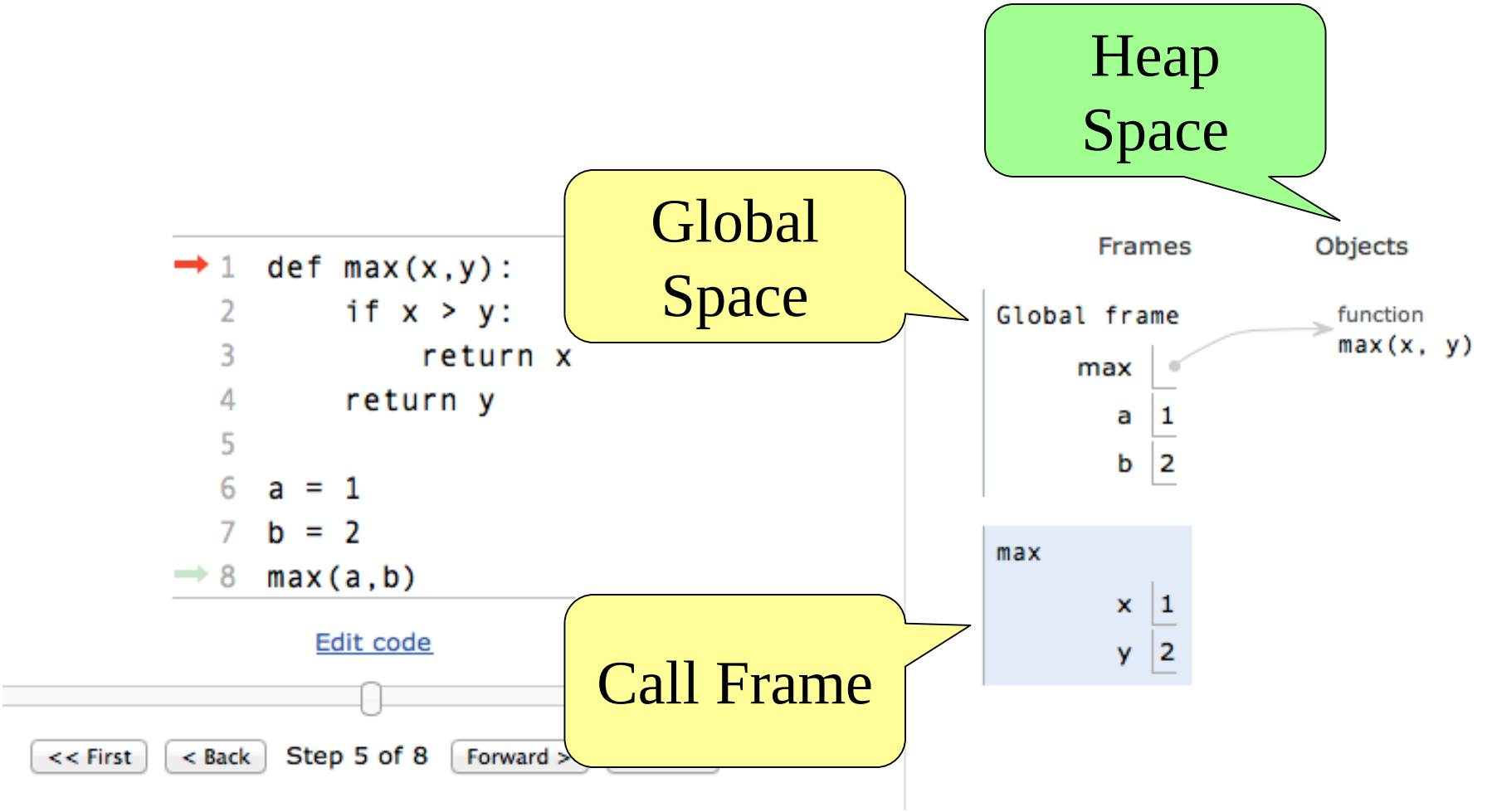
- Assignment 2 is released
  - Due Tuesday, March 7th at 11:59pm
  - Involves writing on *paper*
  - Must turn in a ***legible*** electronic copy through CMS
- Lab 5 is released (note there is no Lab 4)
- Reading: Section 10.1-10.2, 10.4-10.6
- Prelim conflicts assignment on CMS due ***tomorrow*** because **1st Prelim is March 14th**

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
- OPT Link: https://goo.gl/iBfxyo

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

Body

Global Space

to_celsius **id6**

Heap Space

id6

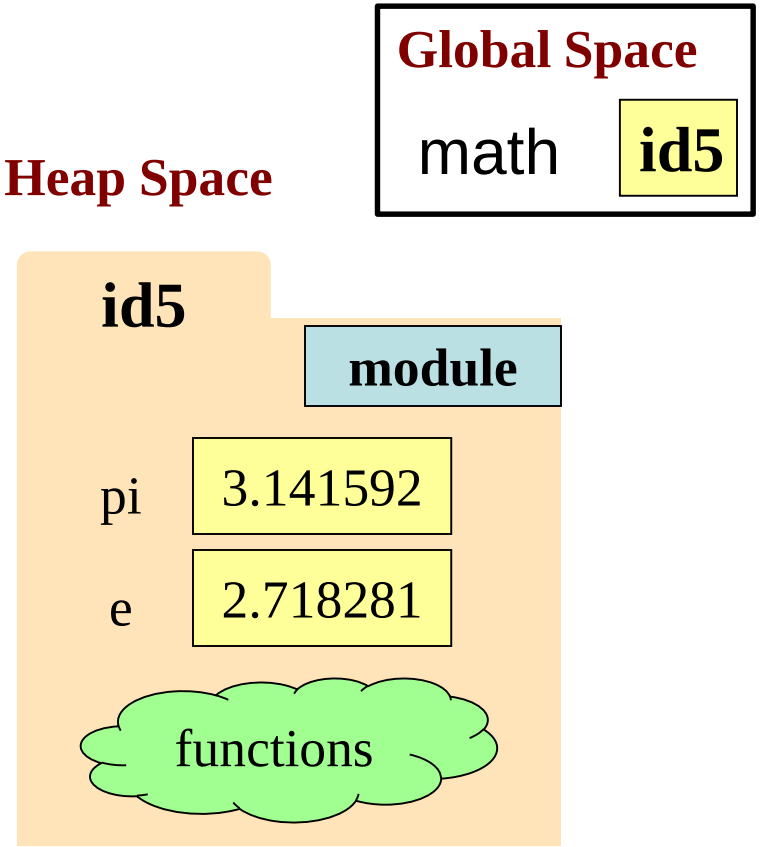
function

Body

Modules and Global Space

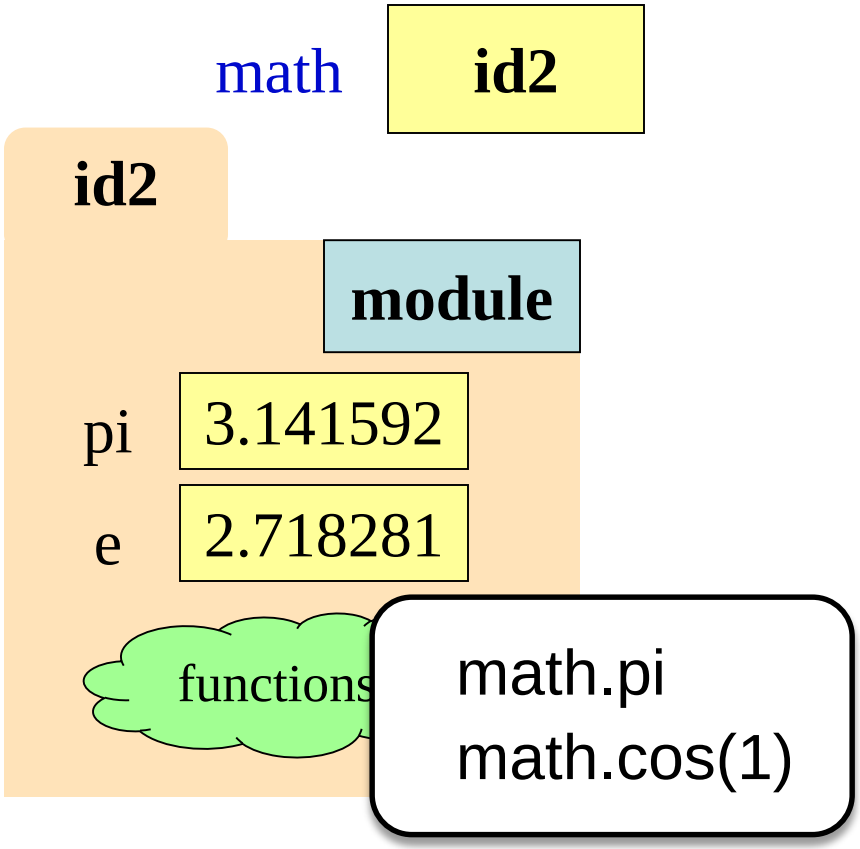
- **import**…
  - Creates a global variable (same name as module)
  - Puts contents in a **folder**
    - variables, functions
  - Puts folder id in variable
- **from** dumps contents to global space
- OPT: https://goo.gl/4LYvwl

import math

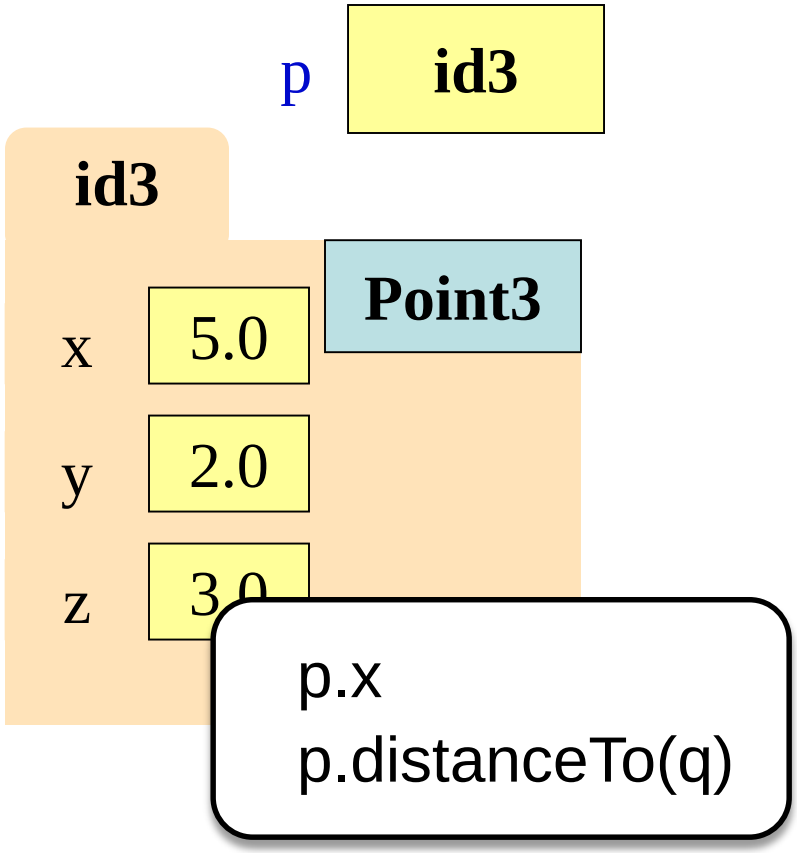


Modules vs Objects

Module



Object



Frames and Helper Functions

- Functions can call each other!
- Each call creates a *new call frame*
- Function that exists mainly to call other functions is often called a **helper function**

From before: last_name_first

```
def last_name_first(n):
    """Returns: copy of <n> but in the form <last-name>, <first-name>

    Precondition: <n> is in the form <first-name> <last-name>
    with one or more blanks between the two names.
    No leading or trailing spaces."""

    space_index = n.find(' ')
    first = n[:space_index]
    last  = n[space_index+1:].strip()
    return last+', '+first
```

- last_name_first('Erik Andersen') gives 'Andersen, Erik'
- last_name_first('Erik Andersen') gives ' Andersen, Erik'

Frames and Helper Functions

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

```
def last_name(s):
    """Prec: see last_name_first"""
    end = s.rfind(' ')
    return s[end+1:]
```

rfind gets the *last* instance of substring

Frames and Helper Functions

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

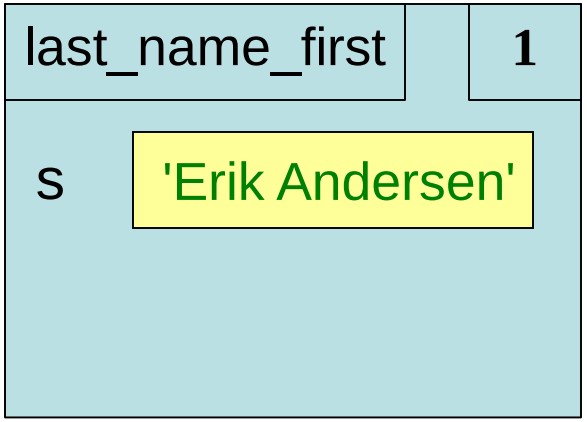
```
def last_name_first(s):
    """Precondition: s in the form
    <first-name> <last-name>"""
    first = first_name(s)
    last = last_name(s)
    return last + ',' + first
```

```
def last_name(s):
    """Prec: see last_name_first"""
    end = s.rfind(' ')
    return s[end+1:]
```

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('Erik Andersen'):



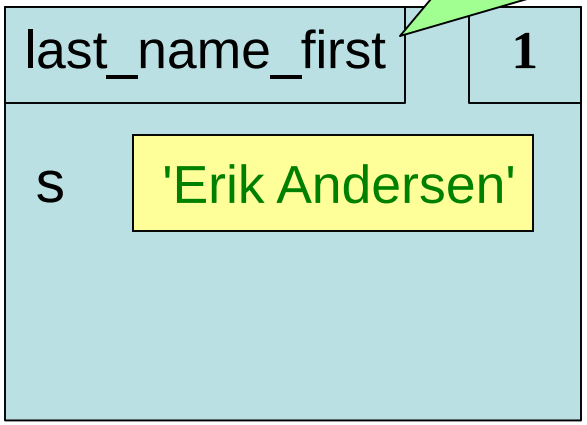

```
def first_name(s):
    """Prec: see last_name_first"""
4   end = s.find(' ')
5   return s[0:end]
```

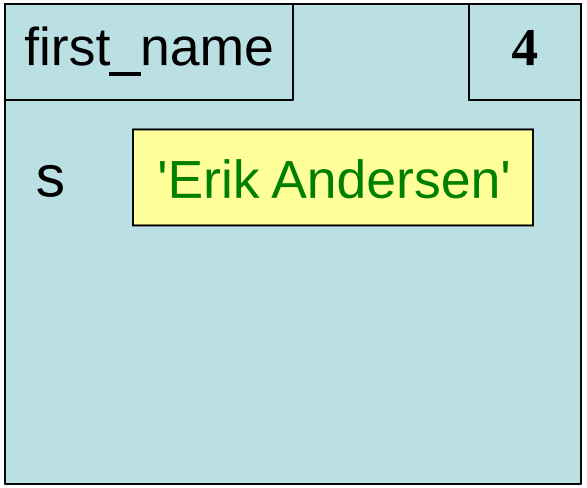
Frames and Helper Functions

```
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"""
4   end = s.find(' ')
5   return s[0:end]
```

Call: last_name_first(...).

Not done. Do not erase!

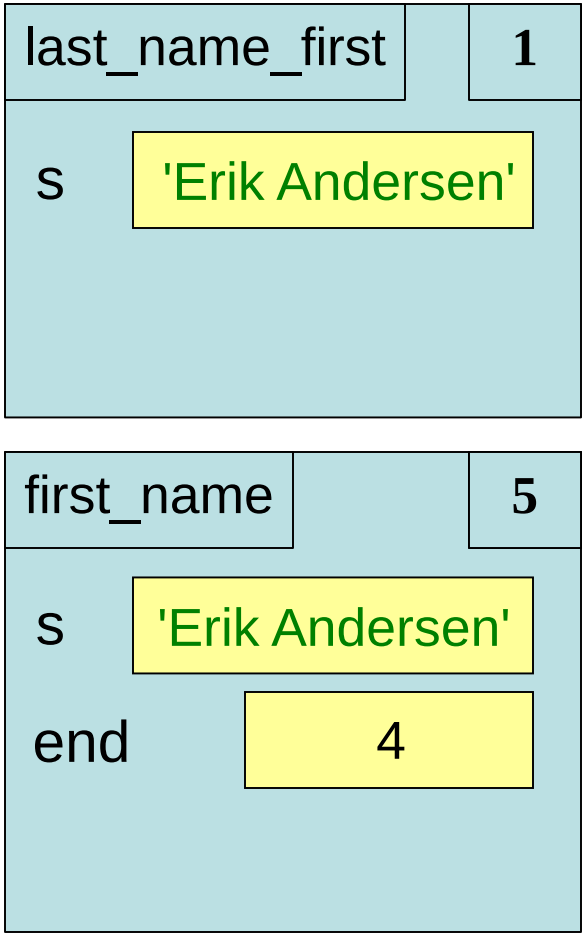




Frames and Helper Functions

```
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"""
4   end = s.find(' ')
5   return s[0:end]
```

Call: last_name_first('Erik Andersen'):

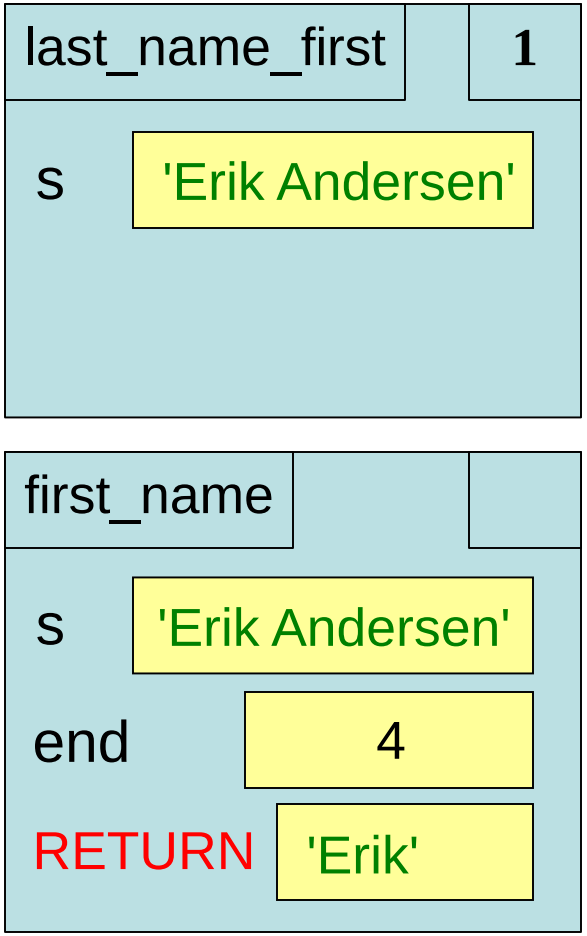


Frames and Helper Functions

```
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"""
4   end = s.find(' ')
5   return s[0:end]
```

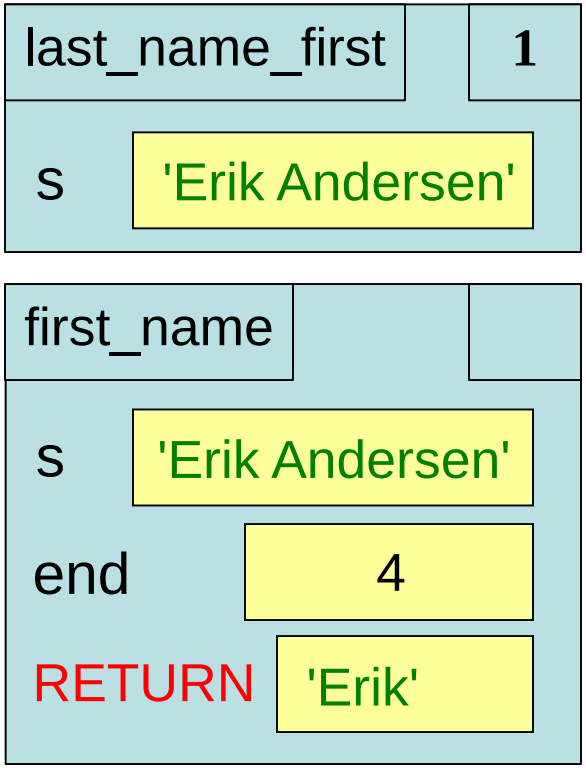
Call: last_name_first('Erik Andersen'):

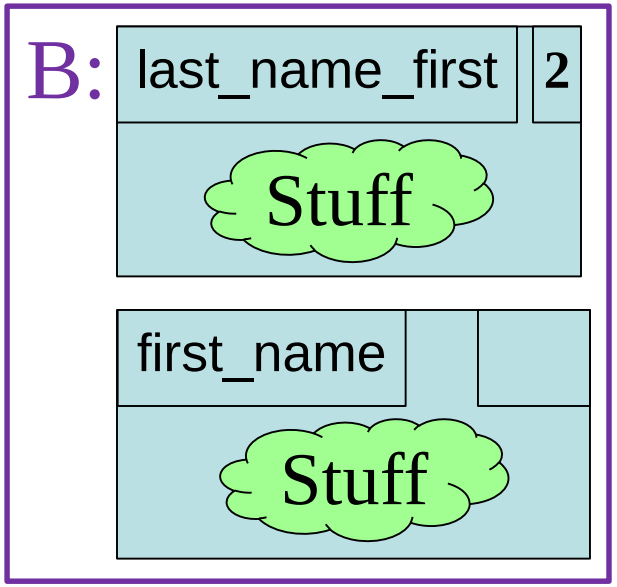


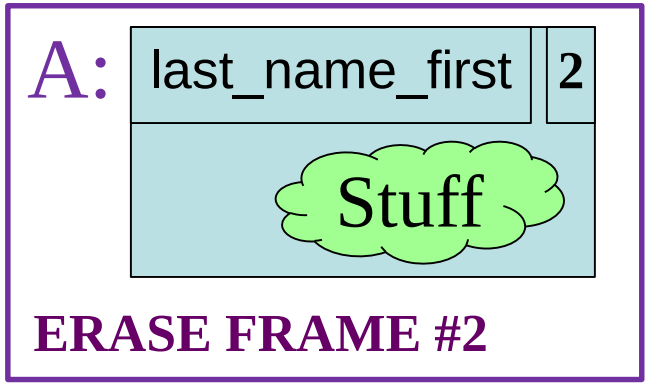
What happens next?

```
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('Erik Andersen'):







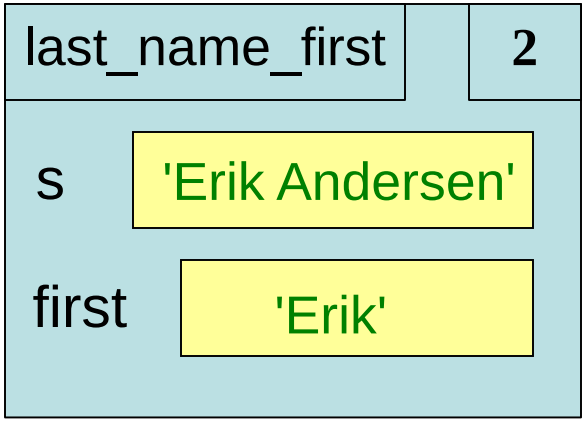
C: **ERASE FRAME #1**

ERASE FRAME #2

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('Erik Andersen'):



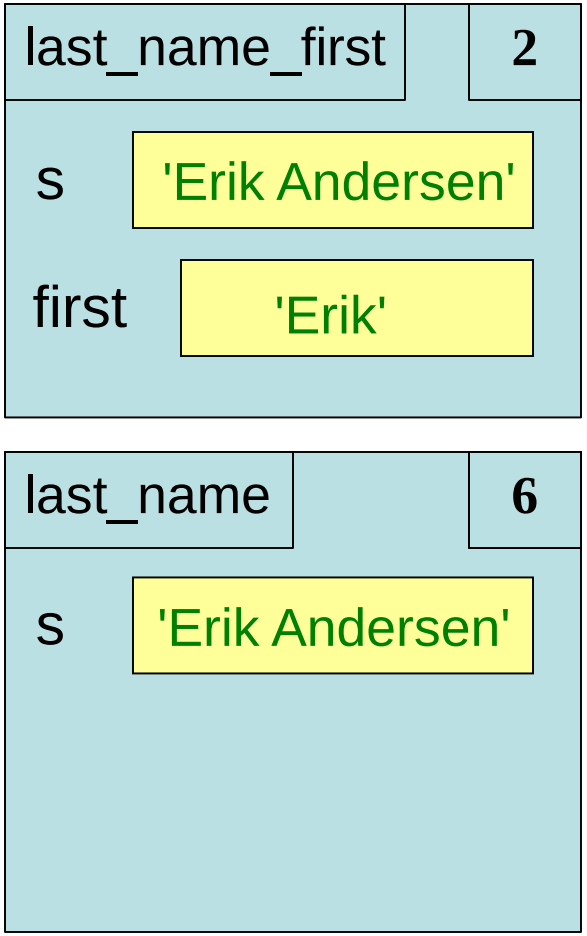

```
def first_name(s):
    """Prec: see last_name_first"""
4   end = s.find(' ')
5   return s[0:end]
```

ERASE WHOLE FRAME

Frames and Helper Functions

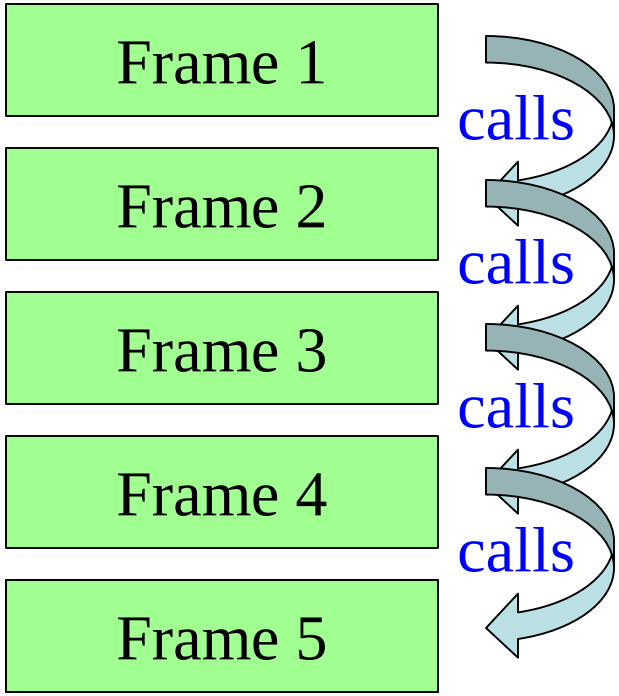
```
def last_name_first(s):
    """Precondition: s in the form
    <first-name> <last-name>"""
1   first = first_name(s)
2   last = last_name(s)
3   return last + '.' + first

def last_name(s):
    """Prec: see last_name_first"""
6   end = s.rfind(' ')
7   return s[end+1:]
```

Call: last_name_first('Erik Andersen'):



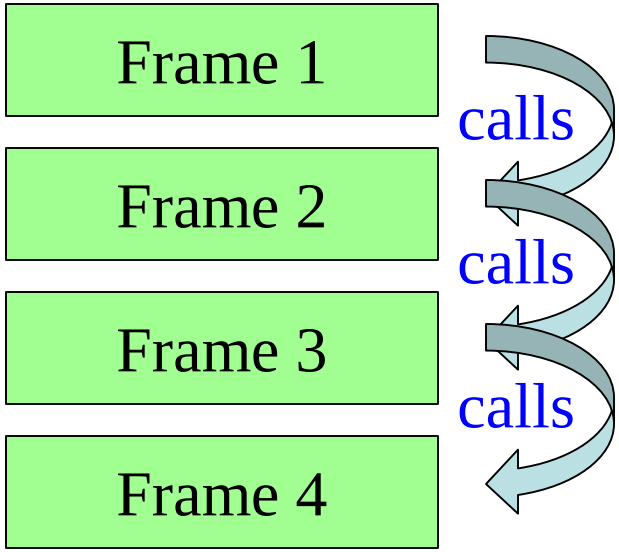
The Call Stack

- Functions are "stacked"
  - Cannot remove one above w/o removing one below
  - Sometimes draw bottom up (better fits the metaphor)
- Python must keep the **entire stack** in memory
  - Error if it cannot hold stack



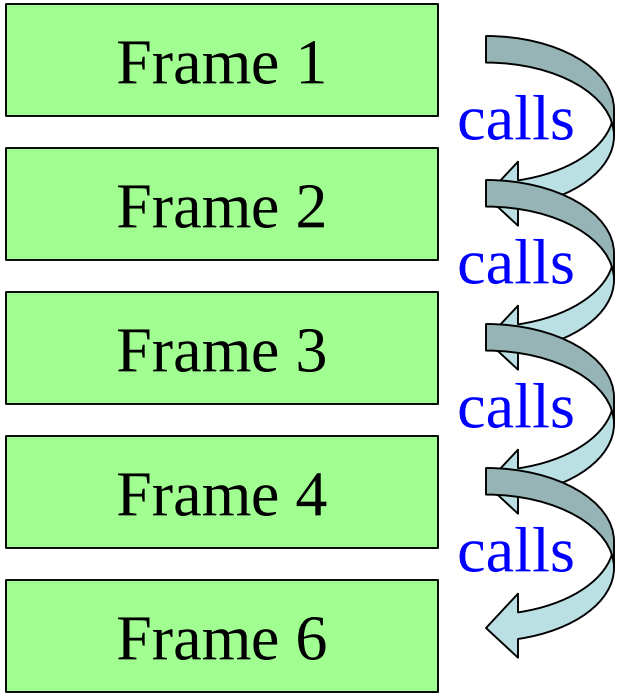
The Call Stack

- Functions are "stacked"
  - Cannot remove one above w/o removing one below
  - Sometimes draw bottom up (better fits the metaphor)
- Python must keep the **entire stack** in memory
  - Error if it 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)
- Python must keep the **entire stack** in memory
  - Error if it cannot hold stack



Example

OPT Link: https://goo.gl/ckBJh9

```
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

print function_1(1,0)
```

calls

calls

calls

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

print function_1(1,0)
```

Crashes here (division by 0)

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

print function_1(1,0)
```

Crashes produce the call stack:

```
Traceback (most recent call last):
  File "error.py", line 20, in <module>
    print function_1(1,0)
  File "error.py", line 7, in function_1
    return function_2(x,y)
  File "error.py", line 11, in function_2
    return function_3(x,y)
  File "error.py", line 15, 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

p
```

Script code.
Global space

Where error occurred
(or where was found)

Crashes produce the call stack:

```
Traceback (most recent call last):
  File "error.py", line 20, in <module>
    print function_1(1,0)
  File "error.py", line 7, in function_1
    return function_2(x,y)
  File "error.py", line 11, in function_2
    return function_3(x,y)
  File "error.py", line 15, in function_3
    return x/y
```

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

assert statement

- **Format**: **assert** *<boolean expression>*
  - Throws error if *<boolean expression>* is False
- **assert** *<boolean expression>, <error message>*
  - Same thing but prints *<error message>*
  - Useful if you want to know what happened

asserting preconditions

- Useful purpose of **assert**: assert preconditions
- Throws error if precondition violated

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

Recovering from Errors

- **try-except** blocks allow us to recover from errors
  - Executes code beneath **try**
  - Once an error occurs, jump to **except**
- **Example**:

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

might have an error

executes if error happens

Comparison

if-else

- **if** vs. **else** depends on Boolean expression
- Never executes both branches

try-except

- Always does **try**
- May not finish **try** if there is an error
  - then goes to **except**

Try-Except is Very Versatile

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

Conversion to a float might fail

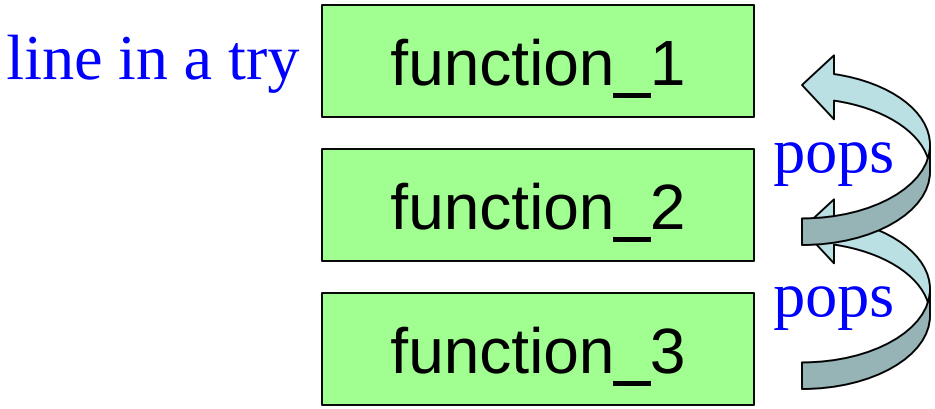
If attempt succeeds, string s is a float

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(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
```

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!

Tracing Control Flow

```
def first(x):
    print 'Starting first.'
    try:
        second(x)
    except:
        print 'Caught at first'
    print 'Ending first'

def second(x):
    print 'Starting second.'
    try:
        third(x)
    except:
        print 'Caught at second'
    print 'Ending second'
```

```
def third(x):
    print 'Starting third.'
    assert x < 1
    print 'Ending third.'
```

What is the output of first(2)?

'Starting first.'
'Starting second.'
'Starting third.'
'Caught at second'
'Ending second'
'Ending first'

Tracing Control Flow

```
def first(x):
    print 'Starting first.'
    try:
        second(x)
    except:
        print 'Caught at first'
    print 'Ending first'

def second(x):
    print 'Starting second.'
    try:
        third(x)
    except:
        print 'Caught at second'
    print 'Ending second'
```

```
def third(x):
    print 'Starting third.'
    assert x < 1
    print 'Ending third.'
```

What is the output of first(0)?

'Starting first.'
'Starting second.'
'Starting third.'
'Ending third'
'Ending second'
'Ending first'