Welcome to CS61A Disc 6 Sect. 29/47 :D

Dickson Tsai

dickson.tsai+cs61a@berkeley.edu

OH: Tu, Th 4-5pm 411 Soda

Previous: Lists and Trees**>>>**

Today: Mutable Lists, Dictionaries, and Nonlocal >>>

Next stop: Object oriented programming

Section Quiz Logistics

- Please take out a piece of paper
- Put your name, login (e.g. cs61a-ty), SID, and section number (#47 for 11-12:30 pm, #29 for 6:30-8pm)
- Graded on effort: effort = revise your quiz with diff color

Section Quiz

1. How can I make your lab/discussion experience more worthwhile?
2. Lab review: Draw out the following data structures.

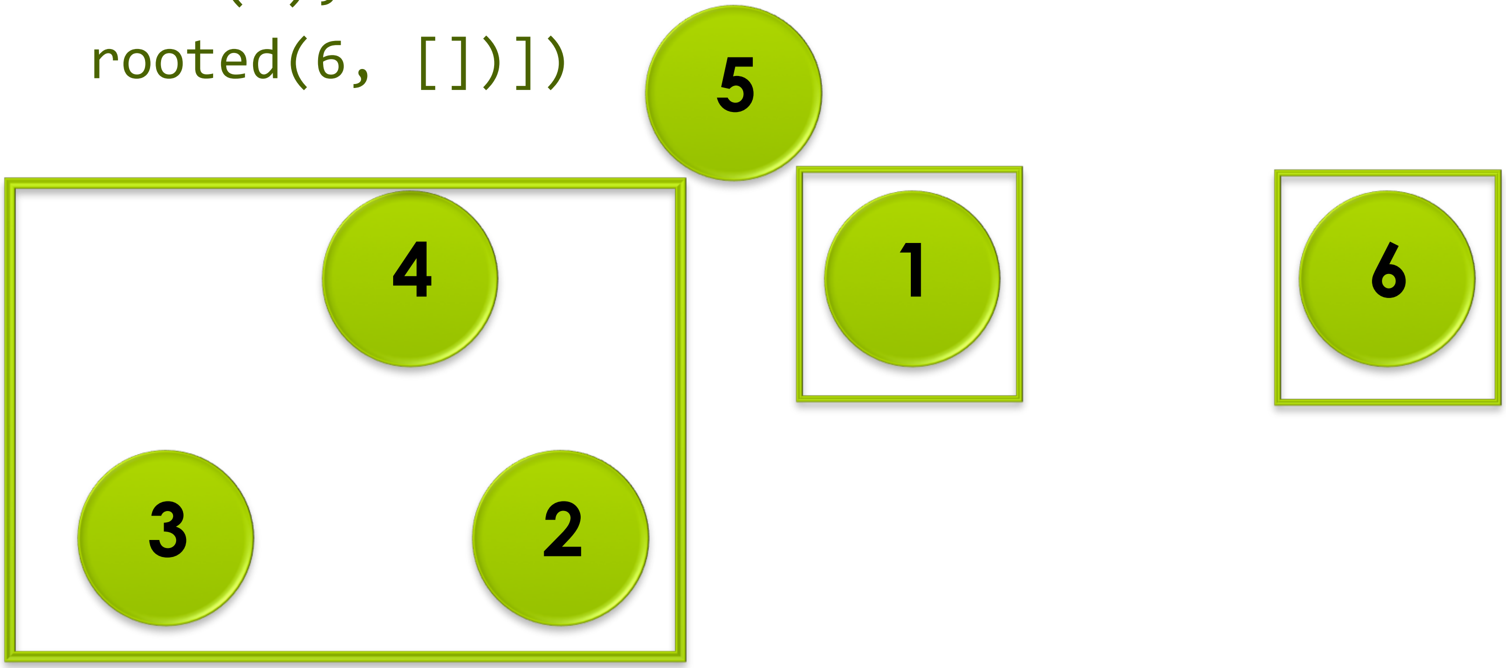
   1.
      ```
      rooted(5, [rooted(4, [leaf(3), leaf(2)]),
            leaf(1), rooted(6, [])])
      ```

   2.
      ```
      link(link(link(‘a’, empty), link(4, empty)),
            link(5, empty))
      ```

   3.
      ```
      reduce(lambda x, y: link(y, x), range(5), empty)
      ```

Section Quiz [Solutions]

```
rooted(5,
    [rooted(4, [leaf(3), leaf(2)]),
    leaf(1),
    rooted(6, [])])
```

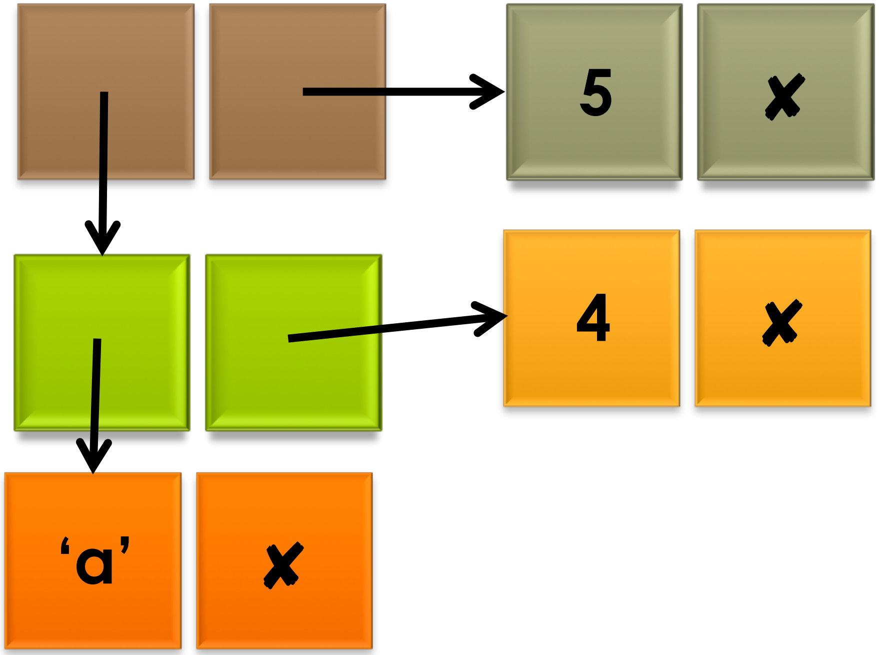


Section Quiz [Solutions]

1st arg can be anything, including another link

```
link(link(link('a', empty), link(4, empty)),
    link(5, empty))
```

2nd arg must always be linked list



Look! 5 link constructor calls, 5 boxes!

Section Quiz [Solutions]

```
reduce(lambda x, y: link(y, x), range(5), empty)
```

x	y	result
empty	0	link(0, empty)
link(0, empty)	1	link(1, link(0, empty))
link(1, link(0, empty))	2	link(2, link(1, link(0, empty)))
link(2, link(1, link(0, empty)))	3	link(3, …)
link(3, …)	4	link(4, link(3, link(2, link(1, link(0, empty)))))

Anuncios

- Project 2 Trends due today for normal credit
- Reminder: My OH are Tuesdays, Thursdays 4-5 PM in Soda 411.
- Do not post any of your code on a public-viewable platform online. (e.g. Github, Pastebin)
- Go to dicksontsai.com/meet to schedule an appointment if you would like to talk to me about anything related to the course.
  - I know office hours are crowded and not individualized. This is my attempt to alleviate that problem.

5-min Review Recap

- Trends project: Why are we dealing w/ ‘sentiment’ ADT??
- Advantages
    - We can now ‘think’ in terms of ‘sentiment’ objects, instead of ‘a number from -1 to 1 or None’.
    - E.g. “““Return a sentiment representing the tweet””” vs. “““Return the value which represents the degree of positivity/ negativity of the tweet, which can range from -1 to 1 or not exist at all (None)”””
- Disadvantages
    - Easy to mess up – abstraction barrier violations only caught by the autograder
    - How would other programmers know that we intend to define an ADT?
- Solution: Make types part of the syntax (next week)

Tip of the Day

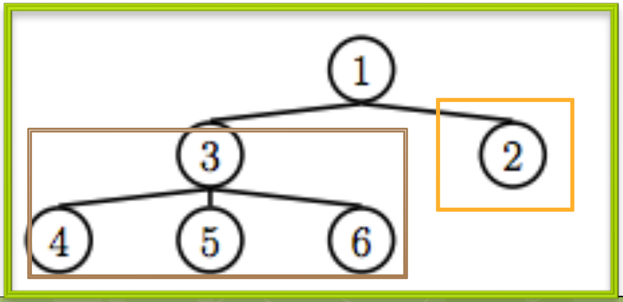
- "An ounce of prevention is worth a pound of cure" – Benjamin Franklin
- Ways to "prevent" trouble later (proven effective from personal experience):
  - Before coding your solution, write pseudocode for it.
    - You want to isolate the conceptual aspect of the problem from the programming/syntax/bugs aspect
    - Example of pseudocode. No Python syntax involved!
      - def sum_squares_list(lst):
        - For each number in the list:
          - Square the number
          - Add the number to a running total
  - Write tests (or at least understand them) before you code

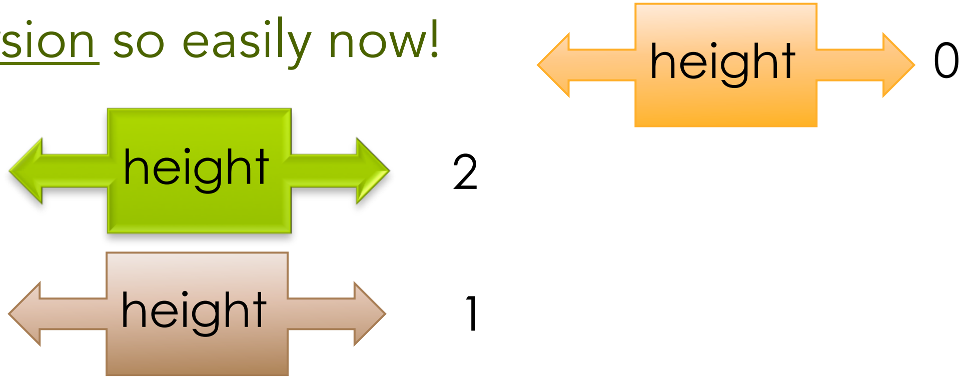
New Way of Running Disc

- Form groups of 2-3 as usual
- <u>Write up your solution (function boxes + answers) on the board!</u>
  - If you are concerned about putting about an incorrect answer, you shouldn’t be. It’s better to make mistakes here than on the test.
    - Plus, everyone can see common mistakes and learn together.
  - More practice with writing code
  - Chalkboards should make you feel smart.
  - Rotate people every problem so everyone gets a chance.

Closure Property

- The result of combination can itself be combined using same method
  - For trees: To create a tree, combine value of list of trees. Now we can put this new tree into another list of trees and value to produce yet another tree!
  - i.e. Every subtree of a tree is a tree
    - -> You can do recursion so easily now!





Mutability

- An object is mutable if it can be changed after it's created
  - Simplest example of mutation you've seen:
    - Assignment!
    - >>> a = 1
    - >>> a = a + 1
  - a's value changed! We did not create a new a!
- Think deeply:
- Q: Are strings mutable? How do you know?
  - Strings are not mutable. We know because they can be keys to a dictionary.

Mutability Powerful but Dangerous

- You can’t make assumptions about a list that’s changing!

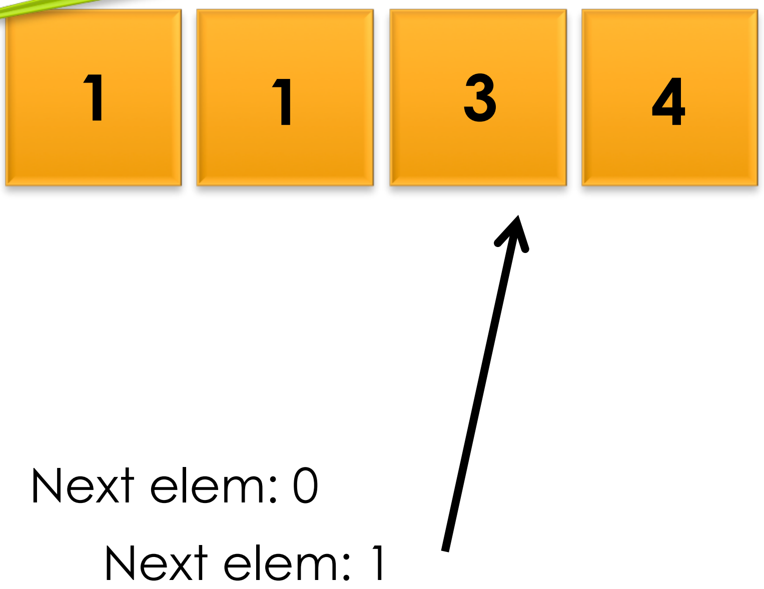
```
>>> def remove_evens(lst):
     for elem in lst:
     if elem % 2 == 0:
      lst.remove(elem)
>>> a = [1, 2, 3, 4]
>>> remove_evens(a)
>>> a
# Do we get [1, 3]?
```

Mutability Powerful but Dangerous

- The list is mutating as you are iterating!

```
>>> def remove_ones(lst):
     for elem in lst:
     if elem == 1:
      lst.remove(elem)
>>> a = [1, 1, 3, 4]
>>> remove_evens(a)
>>> a
# Do we get [3, 4]?
```

The for loop thinks that the next element is at the next index



1	1	3	4

Next elem: 0

Next elem: 1

is vs. ==

- is – an operator that determines if two objects point to the same object in memory
- == - an operator that determines if two objects can be considered ‘equivalent’

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
>>> a == b
True
```

```
>>> a = b = [1, 2, 3]
>>> a is b
True
>>> a == b
True
```

Dictionary

- What are they used for??
  - To organize your data into (key, value) pairs.
  - Especially important if you want to capture general relationships in your data
  - E.g. ‘CA’ <-> list of polygons that represent California
  - More generally:
    - State <-> list of polygons that represent that state
- Keys have to be unique and immutable
- Values can be anything, including other dictionaries

Dictionary – Useful operations

- Create new value
  - Just assign value to a new key

```
>>> a = {}
>>> a['happy'] = 'yes!'
```

- Look up a value
  - Careful! Key must exist

```
>>> a['happy']
'yes!'
```

- Iterate through all keys/values/both

```
>>> for k in a:          >>> for v in a.values()    >>> for k, v in a.items()
…
```

Dictionary – Example

```
>>> poke = {‘p’: 25,
‘d’: 148, ‘m’: 151}
>>> poke[‘p’]
25
>>> poke[‘j’] = 135
>>> poke
{‘p’: 25, ‘d’: 148,
‘m’: 151, ‘j’: 135}
>>> poke[‘di’] = 25
>>> poke
```

Key	Value
‘p’	25
‘d’	148
‘m’	151
‘j’	135
‘di’	25

Nonlocal

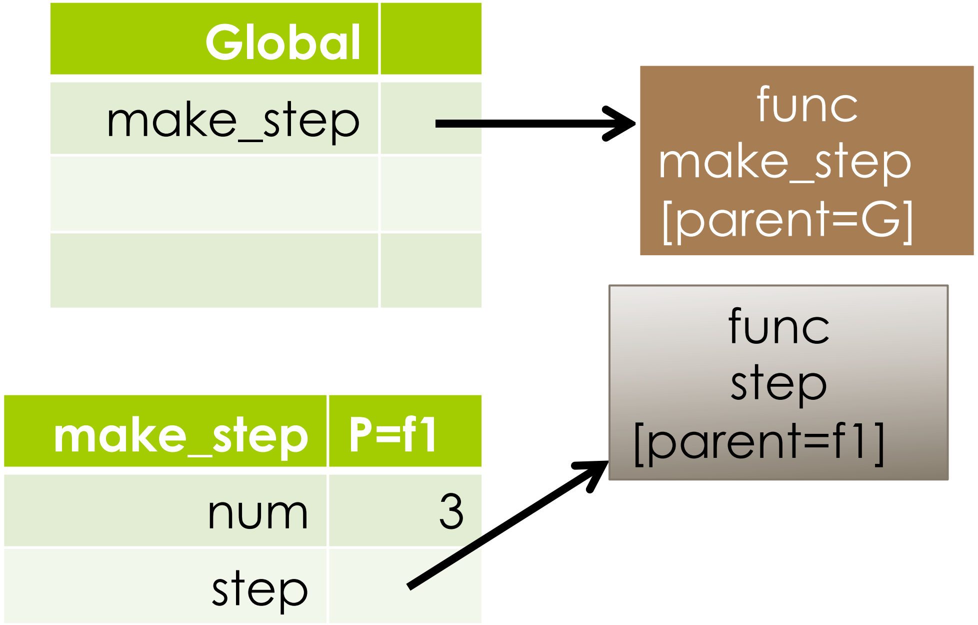
- Why is nonlocal useful??
  - Motivated by the idea of closures: “inner function but not from outside the func”
  - We want the inner function to be able to modify closure variables, not create a local copy for itself. -> Mutation!
- Old assignment rule:
  1. Evaluate the RHS
  2. Bind value(s) from RHS to names on LHS in the current frame.
- The current frame limitation is restricting!

Nonlocal

- Nonlocal modifies assignment rule:
  1. Evaluate the RHS
  2. Bind value(s) from RHS to names on LHS in the enclosing scope (for names declared nonlocal)
- Nonlocal variables: NOT the current frame and NOT the global frame. <u>All frames in between</u> (via parent pointers)

Nonlocal Example

```
def make_step(num):
  def step():
    nonlocal num
    num = num + 1
    return num
  return step
s = make_step(3)
s()
s()
```



Global	
make_step	

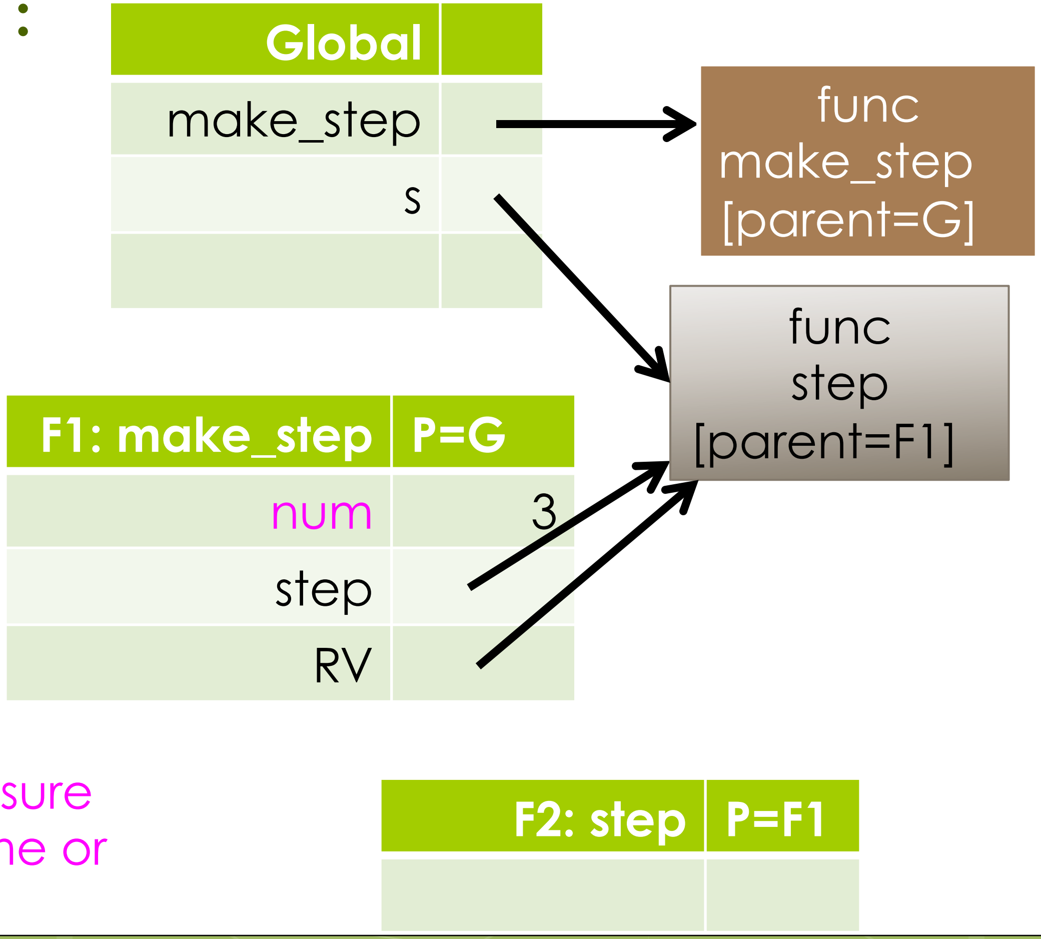
func
make_step
[parent=G]

func
step
[parent=f1]

make_step	P=f1
num	3
step	

Nonlocal Example

```
def make_step(num):
  def step():
    nonlocal num
    num = num + 1
    return num
  return step
s = make_step(3)
s()
s()
```



Global	
make_step	
s	

func
make_step
[parent=G]

func
step
[parent=F1]

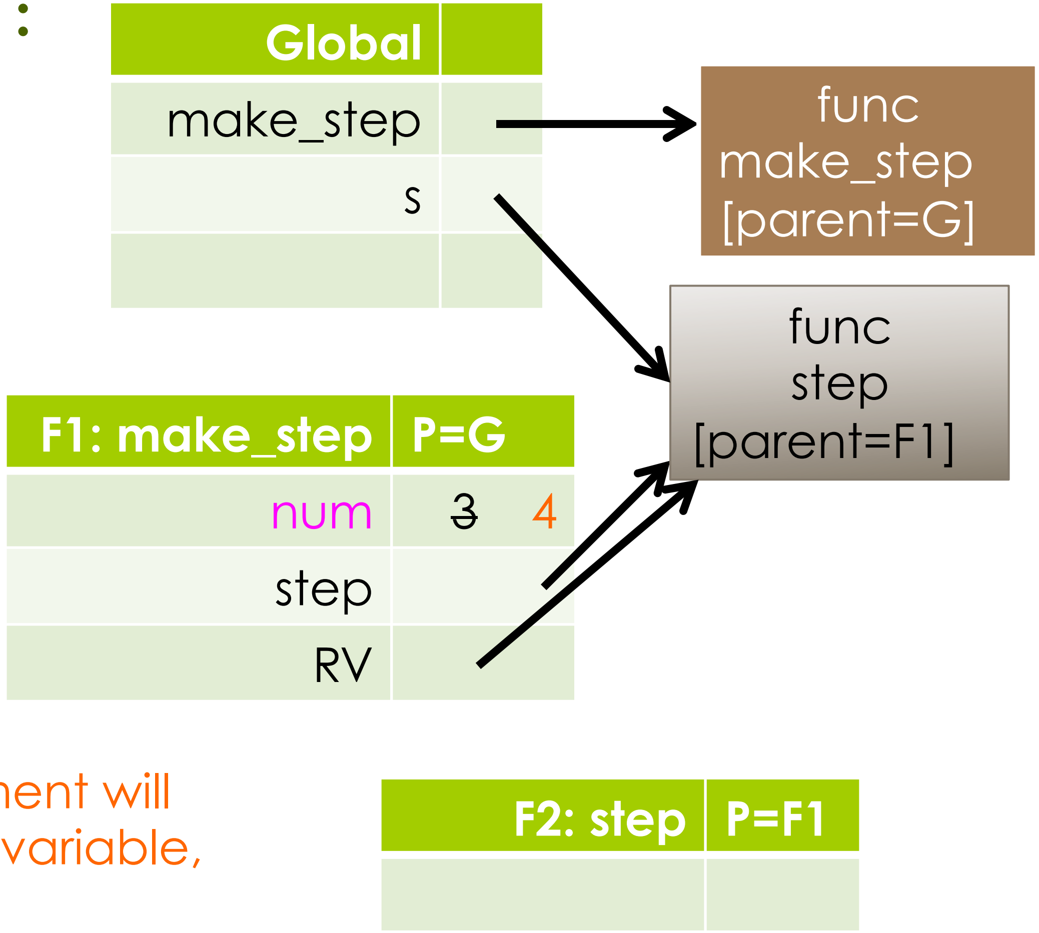
F1: make_step	P=G
num	3
step	
RV	

F2: step	P=F1

Find name in closure (Not current frame or Global).
Name must already exist

Nonlocal Example

```
def make_step(num):
   def step():
      nonlocal num
      num = num + 1
      return num
   return step
s = make_step(3)
s()
s()
```



Global	
make_step	
s	

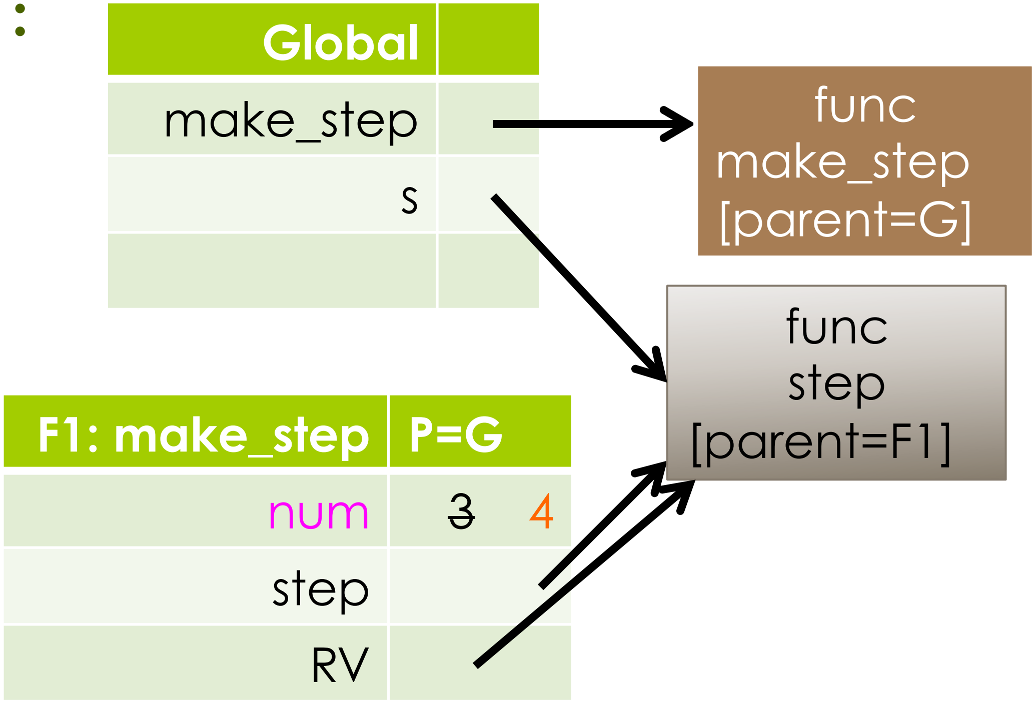
F1: make_step	P=G
num	~~3~~ 4
step	
RV	

F2: step	P=F1

Now the assignment will affect declared variable, not local frame.

Nonlocal Example

```
def make_step(num):
  def step():
    nonlocal num
    num = num + 1
    return num
  return step
s = make_step(3)
s()
s()
```



Global	
make_step	
s	

func
make_step
[parent=G]

func
step
[parent=F1]

F1: make_step	P=G
num	~~3~~ 4
step	
RV	

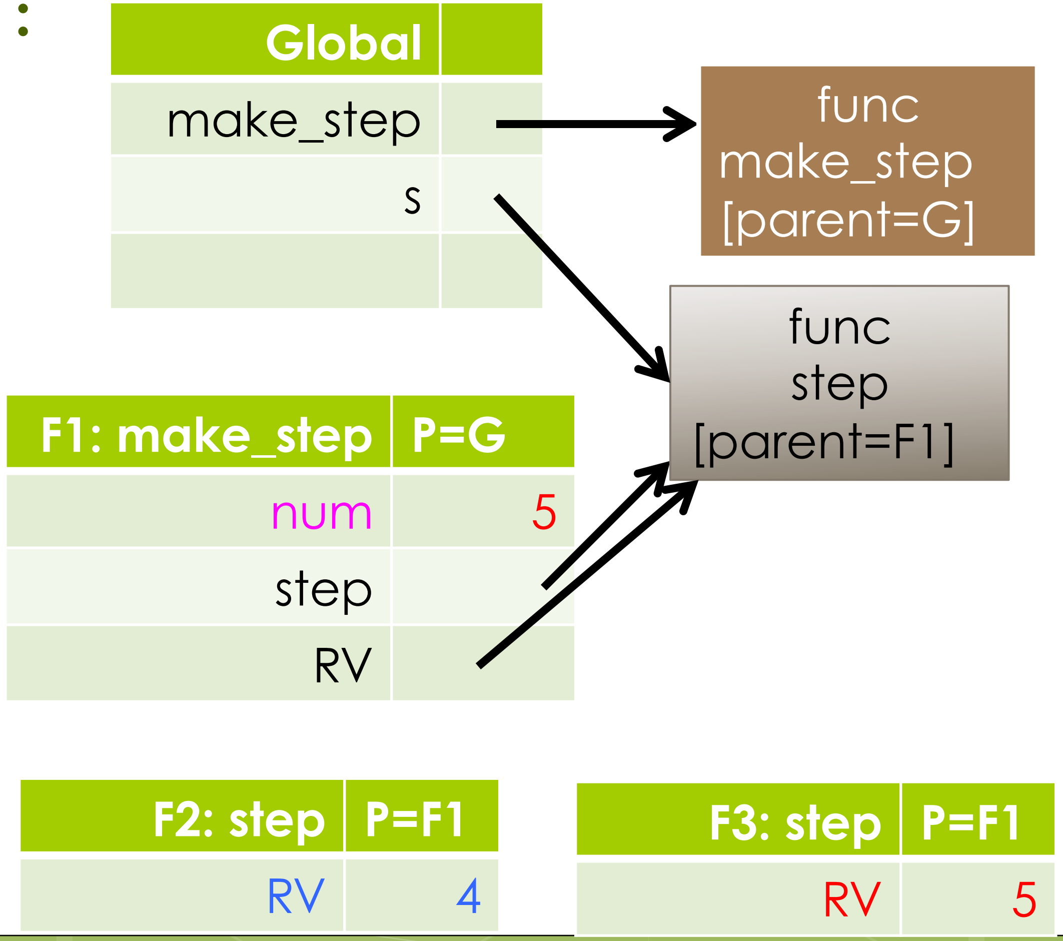
Everything else proceeds as normal.

F2: step	P=F1
RV	4

Nonlocal Example

```
def make_step(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step
s = make_step(3)
s()
s()
```

The next time you call s,
you'll get a different result



Global	
make_step	
s	

func
make_step
[parent=G]

func
step
[parent=F1]

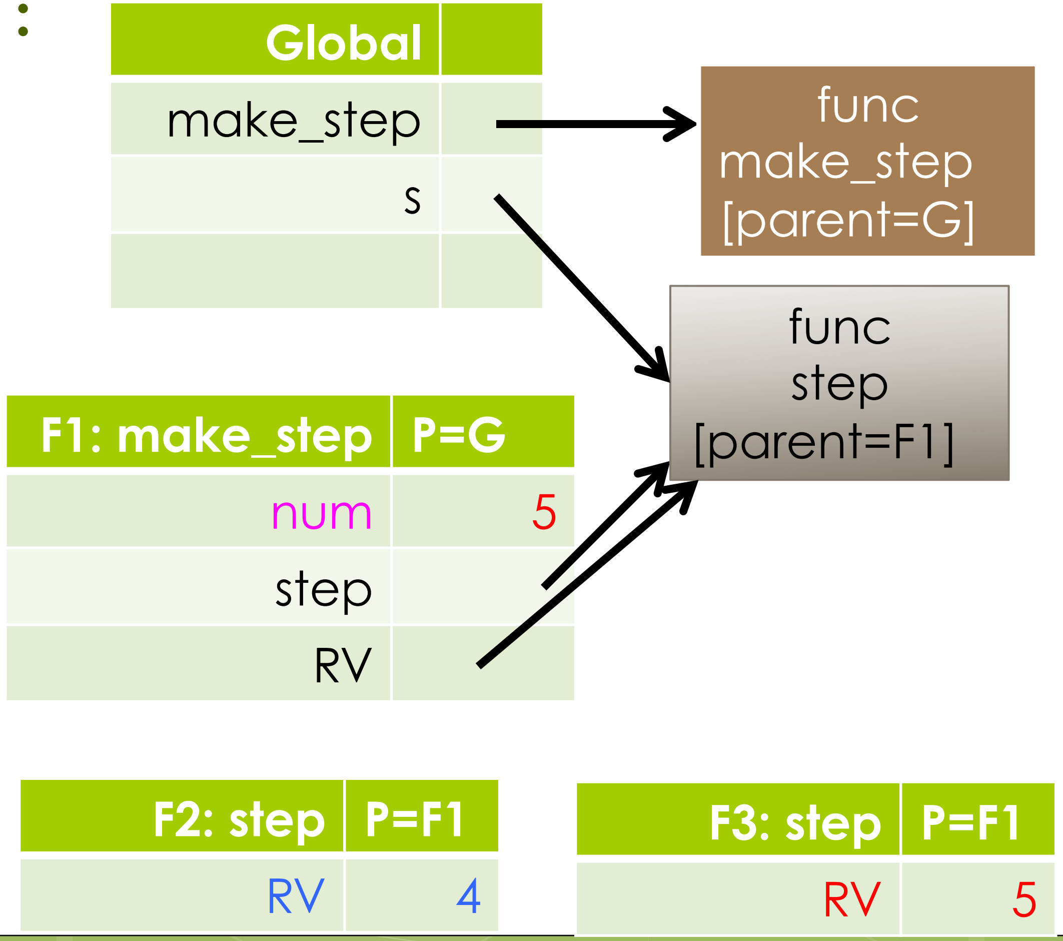
F1: make_step	P=G
num	5
step	
RV	

F2: step	P=F1
RV	4

F3: step	P=F1
RV	5

Nonlocal Example

```
def make_step(num):
  def step():
    nonlocal num
    num = num + 1
    return num
  return step
s = make_step(3)
t = make_step(2)
s()
t()
```



Global	
make_step	
s	

func
make_step
[parent=G]

func
step
[parent=F1]

F1: make_step	P=G
num	5
step	
RV	

F2: step	P=F1
RV	4

F3: step	P=F1
RV	5