Discussion 5:

Trees, Mutation, and Nonlocal

Caroline Lemieux (clemieux@berkeley.edu)

February 28, 2019

Administrativa

Homeworks

HW 4 due tomorrow 3/1

Projects

Maps due today!

Events

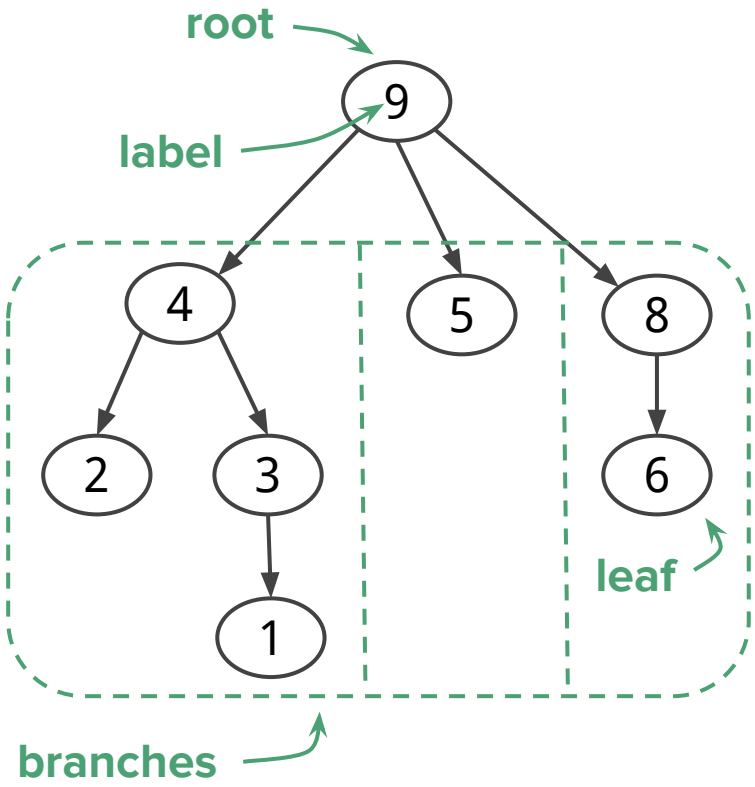
Guerrilla Section Saturday 3/2 12-2PM, Soda 271

Trees

Trees: Terminology

A tree is an ADT

A tree's branches are individual trees themselves.



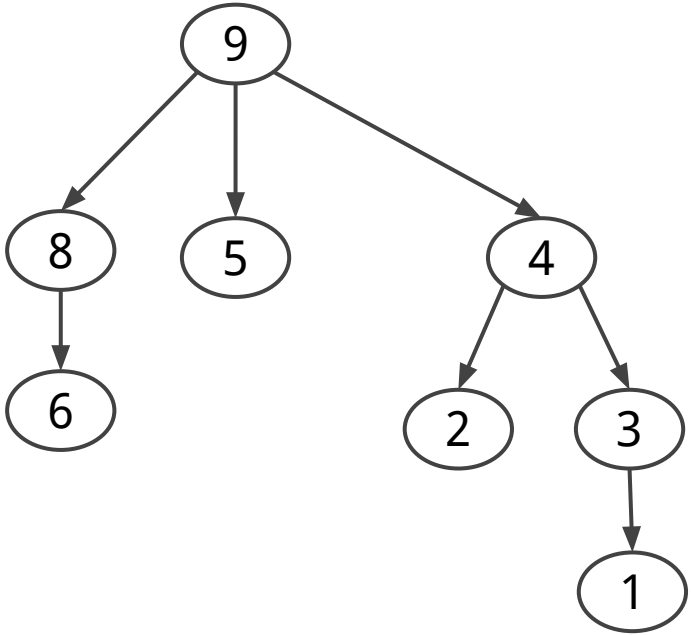
Any confusing terminology on the first page of the discussion?

Trees: What you need to know

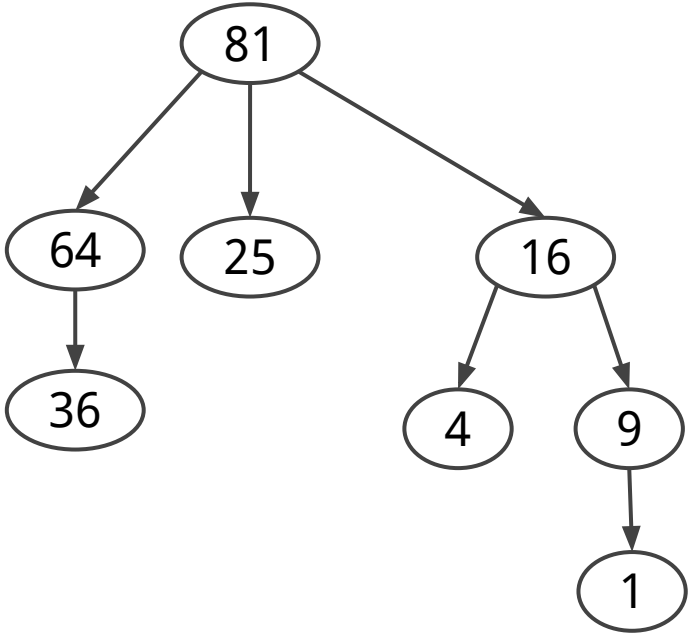
```
tree(label, branches)

label(tree)

branches(tree)

is_leaf(tree)
```

Square Tree (1.2)

Given:

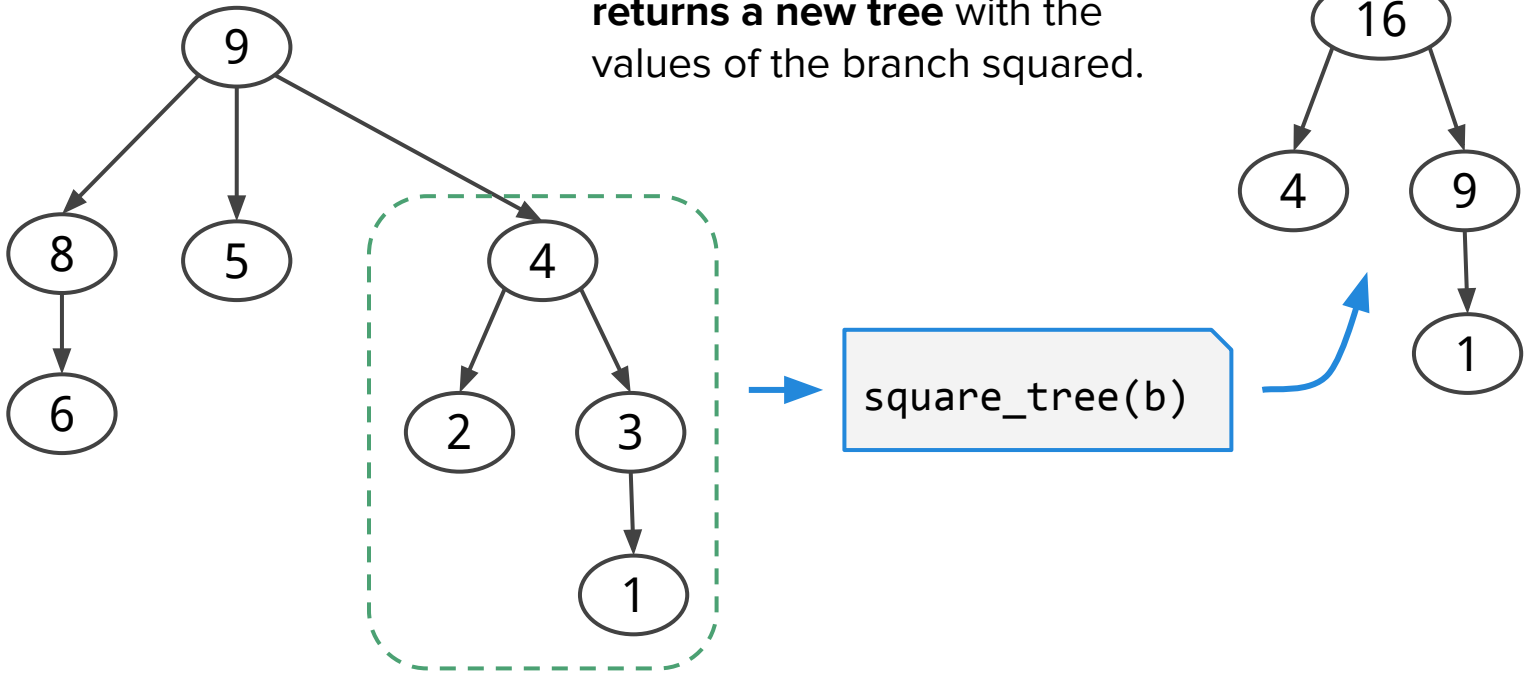


Return:



Square Tree

A recursive call on a branch **returns a new tree** with the values of the branch squared.



Square Tree

```
new_label = label(t) ** 2

new_branches = [square_tree(b) for b in branches(t)]
```

We combine this to get our solution:

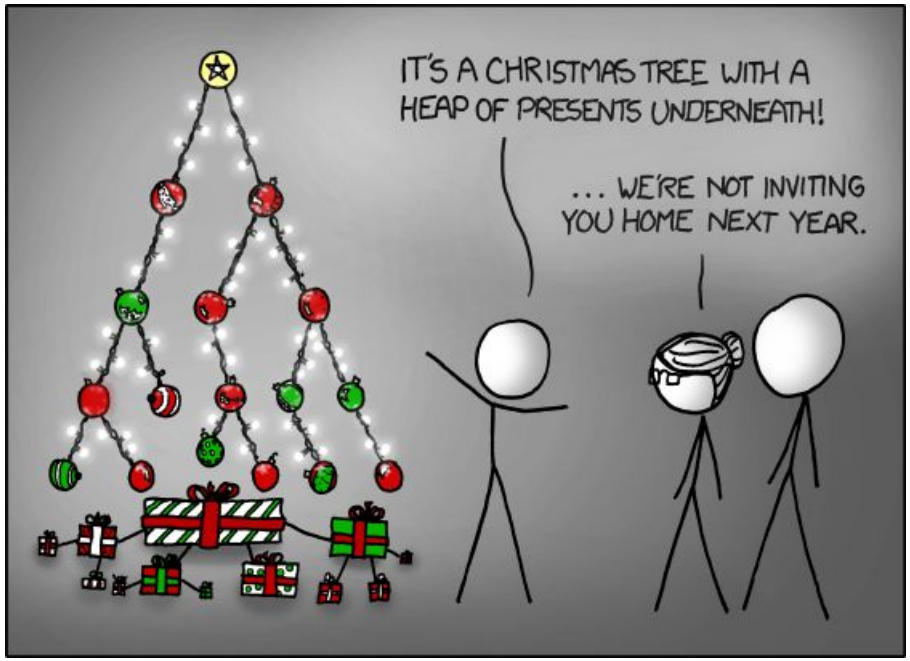
```
def square_tree(t):
    if is_leaf(t):
        return tree(label(t) ** 2)
    new_label = label(t) ** 2
    new_branches = [square_tree(b) for b in branches(t)]
    return tree(new_root, new_branches)
```

Work on 1.1(height) and 1.3(tree_max)!

Try 1.4 if you have time!

Useful functions:

```
tree(label, branches)

label(tree)

branches(tree)

is_leaf(tree)
```



Work on 1.4(find_path)

Attendance

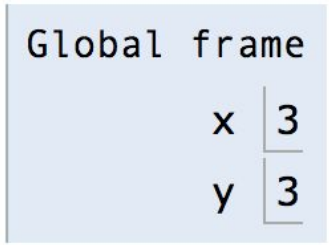
List Mutation

is vs ==

is: is the same thing in the box?

==: are the things in the box equal

```
1  x = 3
2  y = 3
```



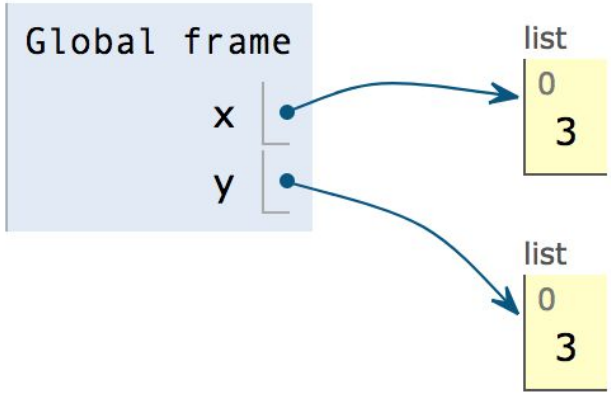

```
>>> x is y
True
>>> x == y
True
```

is vs ==

is: is the same thing in the box?

==: are the things in the box equal

```
1  x = [3]
2  y = [3]
```



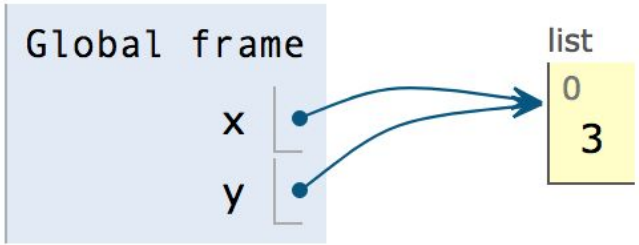

```
>>> x is y
False
>>> x == y
True
```

is vs ==

is: is the same thing in the box?

==: are the things in the box equal

```
1  x = [3]
2  y = x
```

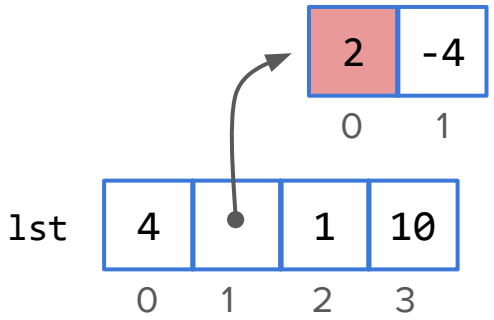



```
>>> x is y
True
>>> x == y
True
```

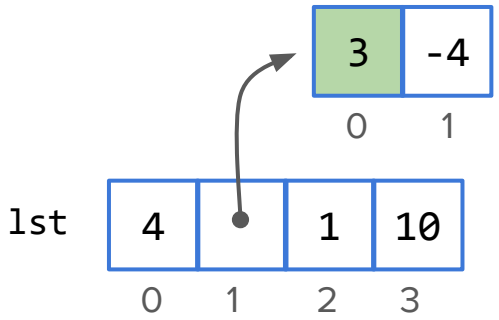
Mutation

A list is a **mutable** object, meaning that we can modify its values!

We use **box-and-pointer diagrams** to keep track of the contents of a list.



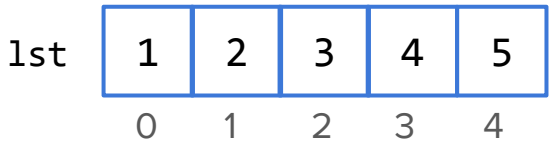

```
lst[1][0] = 3
```



append(el)

Adds `el` to the end of the list.

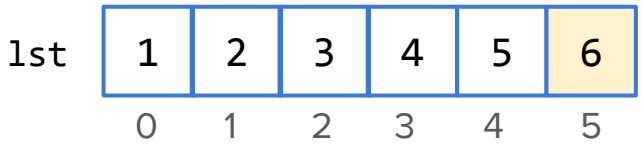
```
>>> lst = [1, 2, 3, 4, 5]
```



append(el)

Adds `el` to the end of the list.

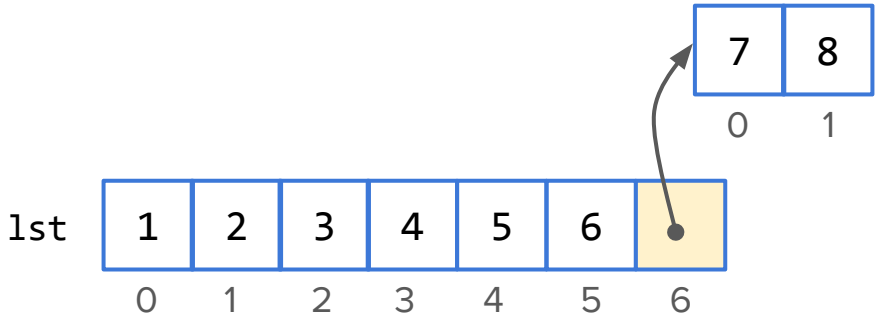
```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.append(6)
```



append(el)

Adds `el` to the end of the list.

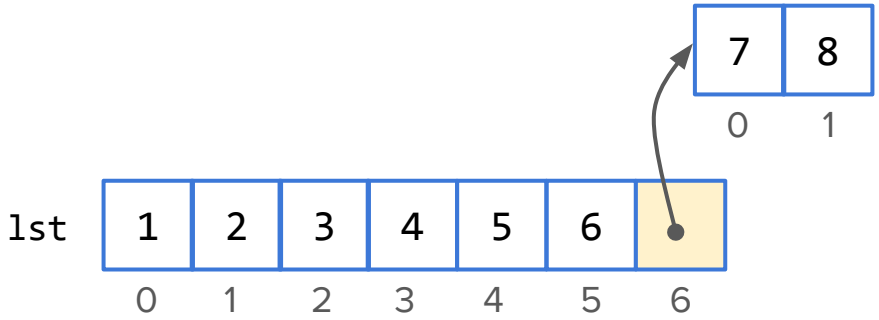
```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.append(6)
>>> lst.append([7, 8])
```



append(el)

Adds `el` to the end of the list.

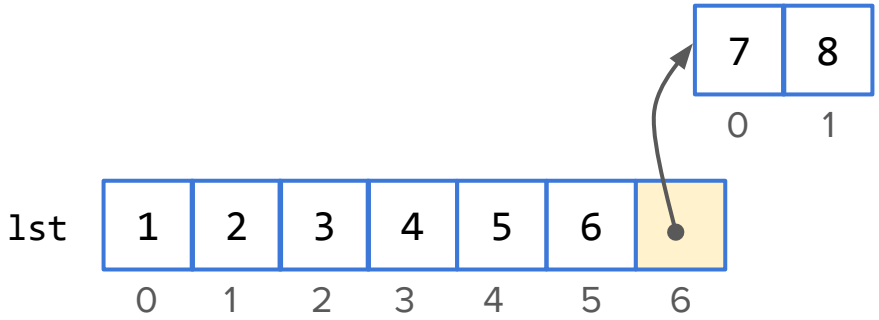
```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.append(6)
>>> lst.append([7, 8])
>>> lst
[1, 2, 3, 4, 5, 6, [7, 8]]
```



append(el)

Adds `el` to the end of the list.

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.append(6)
>>> lst.append([7, 8])
>>> lst
[1, 2, 3, 4, 5, 6, [7, 8]]
```

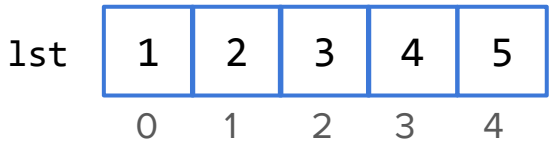


Tip: append adds a *single item* to the list. No matter what, you'll only have to draw one more box at the end of the list.

extend(lst)

Concatenates `lst` to the end of the list.

```
>>> lst = [1, 2, 3, 4, 5]
```



extend(lst)

Concatenates `lst` to the end of the list.

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.extend([6, 7, 8])
```

extend(lst)

Concatenates `lst` to the end of the list.

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.extend([6, 7, 8])
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8]
```

extend(lst)

Concatenates `lst` to the end of the list.

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.extend([6, 7, 8])
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8]
```

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.extend(6)
```

extend(lst)

Concatenates `lst` to the end of the list.

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.extend([6, 7, 8])
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8]
```

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.extend(6)

ERROR
```

extend must be given a *list* (or something else you can iterate through).

extend(lst)

Concatenates `lst` to the end of the list.

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.extend([6, 7, 8])
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8]
```

Tip: there are two ways to think about extend...

- Sticks a list at the end of the original
- Goes through `lst` one item at a time and appends each one

WARNING: extend and +=

Extend does the **same thing** as lst += [...], but is **different** from lst = lst + [...].

```
>>> a = [1, 2, 3]
>>> b = a
>>> a += [4, 5]
>>> b
```

WARNING: extend and +=

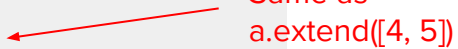
Extend does the **same thing** as lst += [...], but is **different** from lst = lst + [...].

```
>>> a = [1, 2, 3]
>>> b = a
>>> a += [4, 5]
>>> b
[1, 2, 3, 4, 5]
```

WARNING: extend and +=

Extend does the **same thing** as lst += [...], but is **different** from lst = lst + [...].

```
>>> a = [1, 2, 3]
>>> b = a
>>> a += [4, 5]
>>> b
[1, 2, 3, 4, 5]
```



WARNING: extend and +=

Extend does the **same thing** as lst += [...], but is **different** from lst = lst + [...].

```
>>> a = [1, 2, 3]
>>> b = a
>>> a += [4, 5]
>>> b
[1, 2, 3, 4, 5]
```

Same as
a.extend([4, 5])

```
>>> a = [1, 2, 3]
>>> b = a
>>> a = a + [4, 5]
>>> b
```

WARNING: extend and +=

Extend does the **same thing** as lst += [...], but is **different** from lst = lst + [...].

```
>>> a = [1, 2, 3]
>>> b = a
>>> a += [4, 5]
>>> b
[1, 2, 3, 4, 5]
```

Same as a.extend([4, 5])

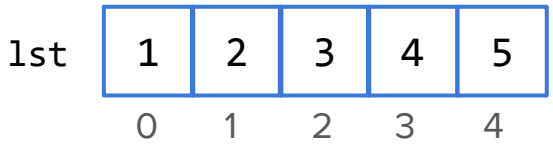
```
>>> a = [1, 2, 3]
>>> b = a
>>> a = a + [4, 5]
>>> b
[1, 2, 3]
```

a now points to an entirely new list! The original was *not* mutated.

insert(i, el)

Inserts `el` at index `i`, shifting the rest of the elements over.

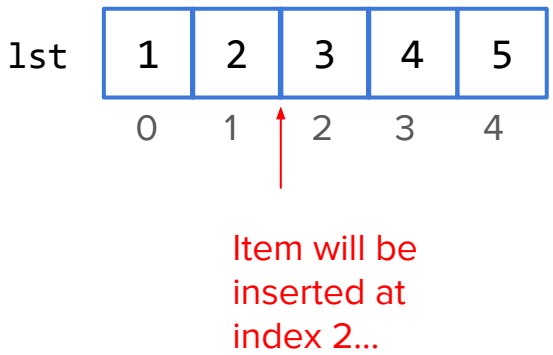
```
>>> lst = [1, 2, 3, 4, 5]
```



insert(i, el)

Inserts `el` at index `i`, shifting the rest of the elements over.

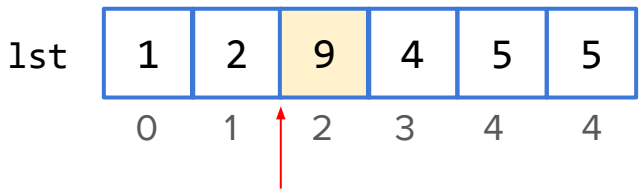
```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.insert(2, 9)
```



insert(i, el)

Inserts `el` at index `i`, shifting the rest of the elements over.

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.insert(2, 9)
```



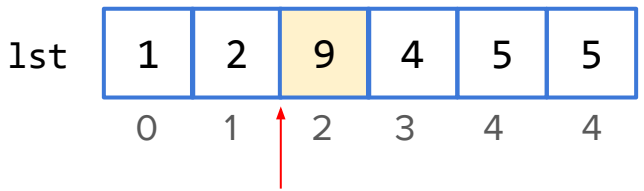
Item inserted at index 2...

rest of list shifted right to make room.

insert(i, el)

Inserts `el` at index `i`, shifting the rest of the elements over.

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.insert(2, 9)
>>> lst
[1, 2, 2.5, 3, 4, 5]
```



Item inserted at index 2...

rest of list shifted right to make room.

Removing items

remove(el): removes the first occurence of `el` from the list

```
>>> lst = [1, “oops”, 3, 4, 5]
>>> lst.remove(“oops”)
>>> lst
[1, 3, 4, 5]
```

pop(i): removes and returns the element at index `i`

```
>>> lst = [1, 2, 3, 4, “hi”]
>>> lst.pop(3)
4
>>> lst.pop() # default: last item
“hi”
>>> lst
[1, 2, 3]
```

NOTE: **remove** takes an *item* to look for and delete.
pop takes an *index*, and returns the item that was deleted as a result.

Check for Understanding

What does the following code display?

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst = lst.append(6)
>>> lst
```

a. [1, 2, 3, 4, 5, 6]
b. [1, 2, 3, 4, 5, [6]]
c. Error
d. None
e. Nothing

Check for Understanding

What does the following code display?

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst = lst.append(6)
>>> lst
```

a. [1, 2, 3, 4, 5, 6]
b. [1, 2, 3, 4, 5, [6]]
c. Error
d. None
e. Nothing ← append returns None, which interpreter doesn't display!

Mutating a list vs. Creating a list

- lst.append(element)
- lst.extend(sequence)
- lst.insert(index, element)
- lst.remove(element)
- lst.pop(index)
- lst1 += lst2

All of above except pop return **None**

- Slicing
  - lst[start:end:step]
- lst1 = lst1 + lst2

Work on 2.1

Draw box and pointer diagrams!!

Work on 2.2

Nonlocal and Mutable Functions

Name lookup

x is found in local frame:

```
def foo():
    x = 10
    def bar(x):
        return x
    return bar

foo()(3)
```

x is found in parent frame:

```
def foo():
    x = 10
    def bar(y):
        return x + y
    return bar

foo()(3)
```

Takeaway: *use binding in current frame if it exists and look in parent frames if it doesn't*

Assignment statements

Assigning a new variable in `bar`:

```
def foo():
    x = 10
    def bar():
        x = 13
        return x
    return bar

foo()()
```

Takeaway: *Assignment statements create/modify new name bindings in the current frame; parent frames are uninvolved*

Nonlocal

By default,

- you *can* **access** variables in parent frames.
- you *cannot* **modify** variables in parent frames.

`nonlocal` statements allow you to **modify** a name in a **parent** frame instead of creating a new binding in the current frame.

- cannot modify variables in current frame
- cannot create bindings in parent frames

```
def foo():
    x = 10
    def bar():
        nonlocal x
        x = 13
    bar()
    return x

foo()
```

This `nonlocal` *statement tells Python: "Don't create a new local variable* x*; modify the one in the parent frame instead!"*

What does this mean?

- We can keep track of things across function calls!
  - Ex: count how many times a function was called, within the function itself
- Functions are not all pure anymore...could have **side effects** (!!!)
  - Could mess with things in other frames
  - Calling the same function twice may give different results
- We've covered all the cases for variable assignment now
  - Referencing variables in **local** and **parent** frames
  - Modifying variables in **local** and **parent** frames

Warmup: WWPD?

```
def g(x):
    def f():
        x = 10
        x = x + 2
    f()
    print(x)
g(20)
```

Warmup: WWPD?

```
def g(x):
    def f():
        x = 10
        x = x + 2
    f()
    print(x)
g(20)
```

20
(local assignment doesn't change parent value)

Warmup: WWPD?

```
def g(x):
    def f():
        x = 10
        x = x + 2
    f()
    print(x)
g(20)
```

20
(local assignment doesn't change parent value)

```
def g(x):
    def f():
        x = 10
        nonlocal x
    f()
    print(x)
g(20)
```

Warmup: WWPD?

```
def g(x):
    def f():
        x = 10
        x = x + 2
    f()
    print(x)
g(20)
```

20
(local assignment doesn't change parent value)

```
def g(x):
    def f():
        x = 10
        nonlocal x
    f()
    print(x)
g(20)
```

Error
(x is used before nonlocal declaration)

Warmup: WWPD?

```
def g(x):
    def f():
        x = 10
        x = x + 2
    f()
    print(x)
g(20)
```

20
(local assignment doesn't change parent value)

```
def g(x):
    def f():
        x = x - 8
    f()
    print(x)
g(20)
```

```
def g(x):
    def f():
        x = 10
        nonlocal x
    f()
    print(x)
g(20)
```

Error
(x is used before nonlocal declaration)

Warmup: WWPD?

```
def g(x):
    def f():
        x = 10
        x = x + 2
    f()
    print(x)
g(20)
```

20
(local assignment doesn't change parent value)

```
def g(x):
    def f():
        x = x - 8
    f()
    print(x)
g(20)
```

Error
(local var 'x' referenced before assignment)

```
def g(x):
    def f():
        x = 10
        nonlocal x
    f()
    print(x)
g(20)
```

Error
(x is used before nonlocal declaration)

Warmup: WWPD?

```
def g(x):
    def f():
        x = 10
        x = x + 2
    f()
    print(x)
g(20)
```

20
(local assignment doesn't change parent value)

```
def g(x):
    def f():
        x = x - 8
    f()
    print(x)
g(20)
```

Error
(local var 'x' referenced before assignment)

```
def g(x):
    def f():
        x = 10
        nonlocal x
    f()
    print(x)
g(20)
```

Error
(x is used before nonlocal declaration)

```
def g(x):
    def f():
        y = 5
        nonlocal x
        x = 10
    f()
    print(x)
g(20)
```

Warmup: WWPD?

```
def g(x):
    def f():
        x = 10
        x = x + 2
    f()
    print(x)
g(20)
```

20
(local assignment doesn't change parent value)

```
def g(x):
    def f():
        x = x - 8
    f()
    print(x)
g(20)
```

Error
(local var 'x' referenced before assignment)

```
def g(x):
    def f():
        x = 10
        nonlocal x
    f()
    print(x)
g(20)
```

Error
(x is used before nonlocal declaration)

```
def g(x):
    def f():
        y = 5
        nonlocal x
        x = 10
    f()
    print(x)
g(20)
```

10
(you can put 'nonlocal' wherever you want, as long as it's before any references!)

Warmup: WWPD?

```
def g(x):
    def f(x):
        nonlocal x
        x = x + 2
    f(1)
    print(x)
g(20)
```

Error

‘x’ is both a parameter (local) and nonlocal. Python doesn’t know which to use!

General nonlocal rules

- Variable declared nonlocal must...
  - Be present in a parent frame
  - Not be in the global frame
  - Not have been declared locally in the current frame (either in the body or as a parameter)