TUPLES, LISTS, MUTABILITY

(download slides and .py files to follow along)

6.0001 LECTURE 5

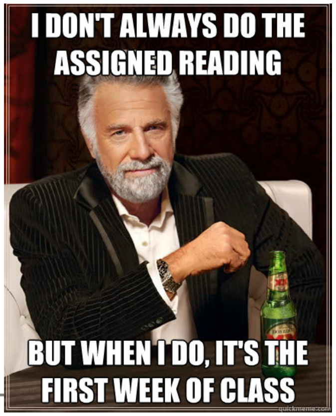
Eric Grimson

LAST FEW LECTURES

- while loops & for loops
  - should know how to write both kinds
  - should know when to use them
    - computations characterized by "state variables" and update rules
- functions
- recursion
- decomposition and abstraction

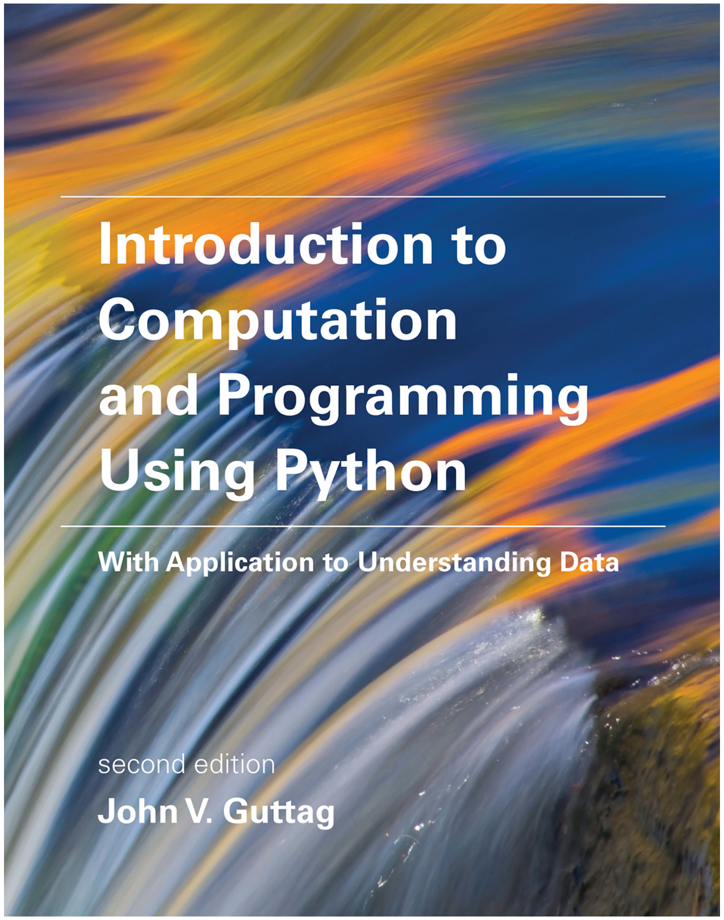
TODAY

- new data types – tuples and lists
- mutable and immutable data structures
  - challenges of aliasing under mutation
- looping or recursing over compound data structures



Assigned Reading

- today:
  - section 5.1 – 5.5
- next lecture:
  - section 5.6
  - chapter 6
  - chapter 7



See https://mitpress.mit.edu/books/introduction-computation-and-programming-using-python-second-edition for errata sheet

A new data type

- Have seen scalar types: `int, float, bool, string`
- Want to introduce new **compound data types**
  - indexed sequences of elements, which could themselves be compound structures
  - tuples – immutable
  - lists – mutable
- Explore ideas of
  - Mutability
  - Aliasing
  - Cloning

TUPLES

- **Indexable ordered sequence** of objects
  - Objects can be any type – int, string, tuple, tuple of tuples, …

Remember strings?

- Cannot change element values, **immutable**

```
te = ()
```
Empty tuple

```
ts = (2,)
```
Extra comma means tuple with one element

Compare with `ts = (2)`

```
t = (2, "mit", 3)
```

`t[0]` → evaluates to 2 — Indexing starts at 0

`(2,"mit",3) + (5,6)` → evaluates to `(2,"mit",3,5,6)`

`t[1:2]` → slice tuple, evaluates to `("mit",)`

`t[1:3]` → slice tuple, evaluates to `("mit",3)`

`len(t)` → evaluates to 3

`max((3,5,0))` → evaluates 5

`t[1] = 4` → gives error, **can't modify object**

INDICES AND SLICING

```
seq = (2,'a',4,(1,2))
```

index: 0 1 2 3

```
print(len(seq))        → 4
print(seq[3])          → (1,2)
print(seq[-1])         → (1,2)
print(seq[3][0])       → 1
print(seq[4])          → error
```

*An element of a sequence is at an **index**, indices start at 0*

```
print(seq[1])          → a
print(seq[-2:]         → (4,(1,2))
print(seq[1:4:2]       → ('a',(1,2))
print(seq[:-1])        → (2,'a',4)
print(seq[1:3])        → 'a',4
```

Slices extract subsequences

```
for e in seq:          → 2
    print(e)             'a'
                         4
                         (1,2)
```

Iterating over sequences

TUPLES

- Conveniently used to **swap** variable values

```
x = y
y = x
```
✗

```
temp = x
x = y
y = temp
```
✓

```
(x, y) = (y, x)
```
✓

- Used to **return more than one value** from a function

```
def quotient_and_remainder(x, y):
    q = x // y
    r = x % y
    return (q, r)
(quot, rem) = quotient_and_remainder(4,5)
both = quotient_and_remainder(4,5)
```

YOUR TURN

Consider the following code:

```
def always_sunny(t1, t2):
    """t1, t2 are non-empty"""
    sun = ("sunny", "sun")
    first = t1[0] + t2[0]
    return (sun[0], first)
```

To what does `always_sunny((‘cloudy’), (‘cold’,))` evaluate?

A) `(‘sunny’, ‘cc’)`

B) `(‘sunny’, ‘ccold’)`

C) `(‘sunny’, ‘cloudycold’)`

D) nothing, it will show an error

LISTS

- **Indexable ordered sequence** of objects
  - Usually homogeneous (i.e., all integers, all strings, all lists)
  - But can contain mixed types (not common)
- Denoted by **square brackets**, [ ]
- **Mutable**, this means you can change element values

Three programmers walk into a bar...

INDICES AND ORDERING

```
a_list = []
L = [2, 'a', 4, [1,2]]
```

empty list

`len(L)` → evaluates to 4 — Gives length of top level of tuple

`L[0]` → evaluates to 2 — Indexing starts at 0

`L[2]+1` → evaluates to 5

`L[3]` → evaluates to `[1,2]`, another list!

`L[4]` → gives an error

`i = 2`
`L[i-1]` → evaluates to `'a'` since `L[1]='a'`

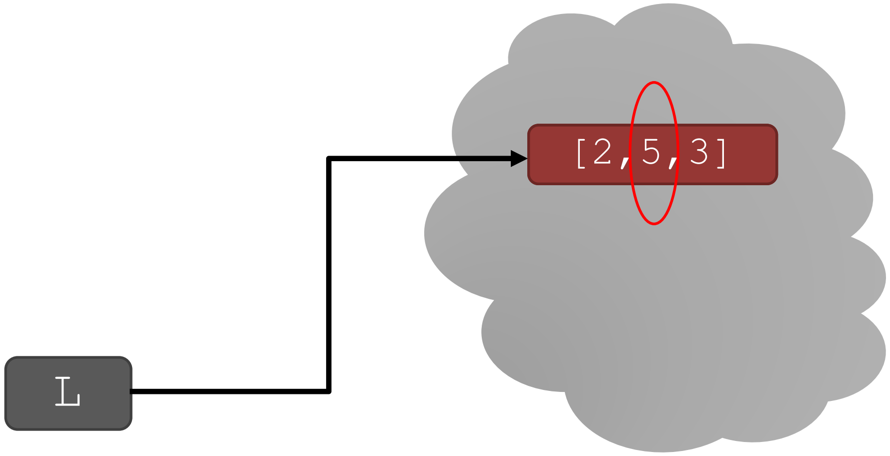
`max([3,5,0])` → evaluates 5

MUTABILITY

- Lists are **mutable**!
- Assigning to an element at an index **changes** the value

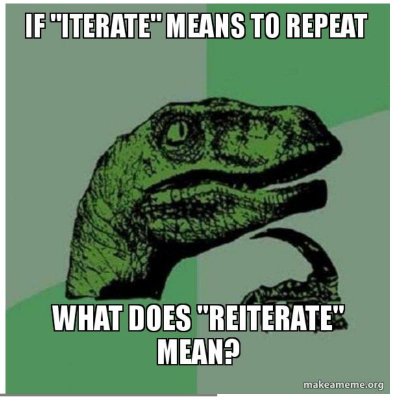
```
L = [2, 1, 3]
L[1] = 5
```

- `L` is now `[2, 5, 3]`; note this is the **same object** `L`



different from strings and tuples!

ITERATING OVER A LIST



- Compute the **sum of elements** of a list
- Common pattern

```
total = 0
for i in range(len(L)):
    total += L[i]
print(total)
```

```
total = 0
for i in L:
    total += i
print(total)
```

Like strings, can iterate over list elements **directly**

This version is more "pythonic"!

- Notice
  - List elements are indexed `0` to `len(L)-1`
  - `range(n)` goes from `0` to `n-1`

YOUR TURN


```
L= ["life", "answer", 42, 0]

for thing in L:
    if thing == 0:
        L[thing] = "universe"
    elif thing == 42:
        L[1] = "everything"
```

What is the value of L after you run this code?

A) `["life", "answer", 42, 0]`

B) `["universe", "answer", 42, 0]`

C) `["universe", "everything", 42, 0]`

D) `["life", "everything", 42, 0]`

OPERATION ON LISTS: append

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```

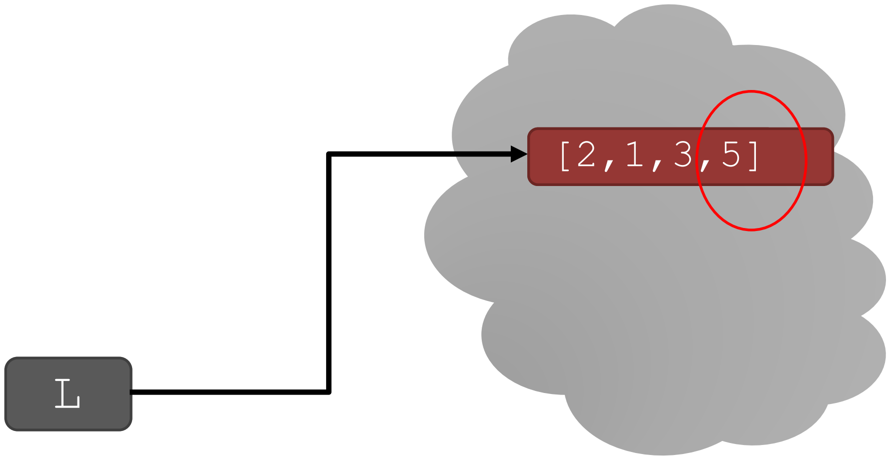
→ L is now `[2,1,3,5]`

- What is the dot?
  - Lists are Python objects, everything in Python is an object
  - Objects have data
  - Objects have methods and functions
  - Access this information by `object_name.do_something()`
  - Equivalent to calling `append` with arguments `L` and `5`
  - Will learn more about these later

OPERATION ON LISTS – append

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`

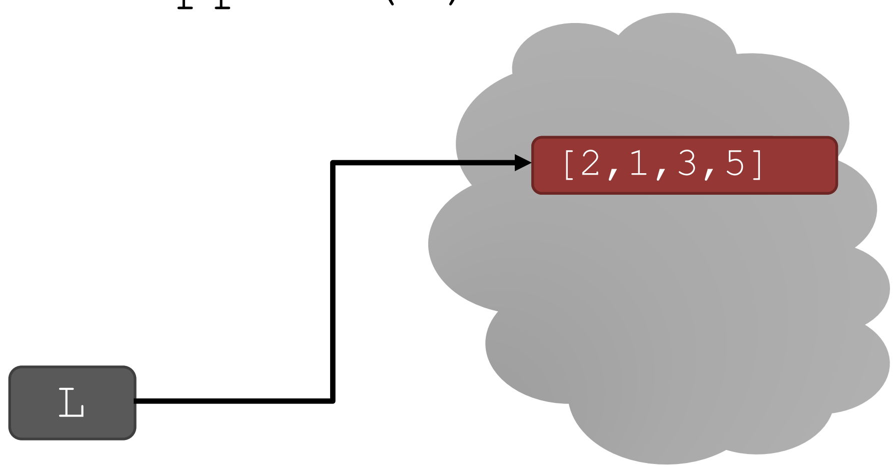


OPERATION ON LISTS – append

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`

```
L = L.append(5)
```

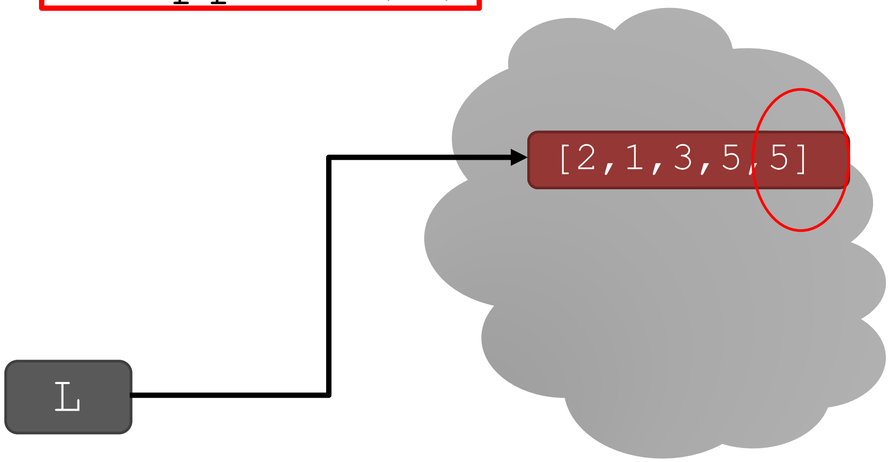


OPERATION ON LISTS – append

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`

```
L = L.append(5)
```

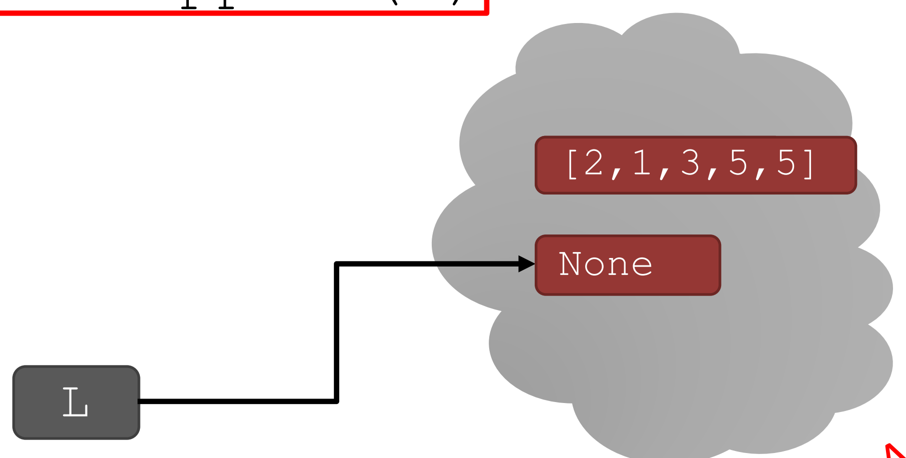


OPERATION ON LISTS – append

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
L = L.append(5)
```

→ L is now `[2,1,3,5]`



Be careful! The append operation does a mutation, but returns the None object as a result.

Append is used strictly for its **side effect**

TRICKY EXAMPLE 1: append

- **Range returns something that behaves like a tuple** (but isn't – it returns an *iterable*)
- Returns the first element, and an iteration method by which subsequent elements are generated as needed

`range(4)` → equivalent to tuple `(0,1,2,3)`
`range(2,6)` → equivalent to tuple `(2,3,4,5)`

```
L = [1,2,3,4]
for i in range(len(L)):
    L.append(i)
    print(L)
```

Iteration sequence is pre-determined at beginning of loop

0th time: L is [1, 2, 3, 4, 0]

1st time: L is [1, 2, 3, 4, 0, 1]

2nd time: L is [1, 2, 3, 4, 0, 1, 2]

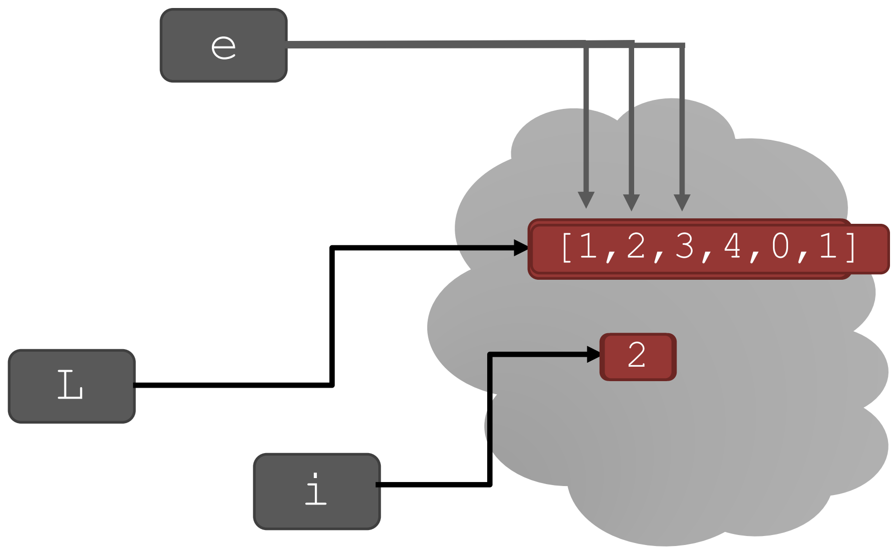
3rd time: L is [1, 2, 3, 4, 0, 1, 2, 3]

TRICKY EXAMPLE 2: append

```
L = [1,2,3,4]
i = 0
for e in L:
    L.append(i)
    i += 1
    print(L)
```

Originally [1,2,3,4]

L is **mutated** each iteration



0th time: L is [1, 2, 3, 4, 0]

1st time: L is [1, 2, 3, 4, 0, 1]

2nd time: L is [1, 2, 3, 4, 0, 1, 2]

3rd time: L is [1, 2, 3, 4, 0, 1, 2, 3]

NEVER STOPS!

In previous example, L was accessed at onset to create a range iterable; in this example, the loop is directly accessing indices into L

COMBINING LISTS

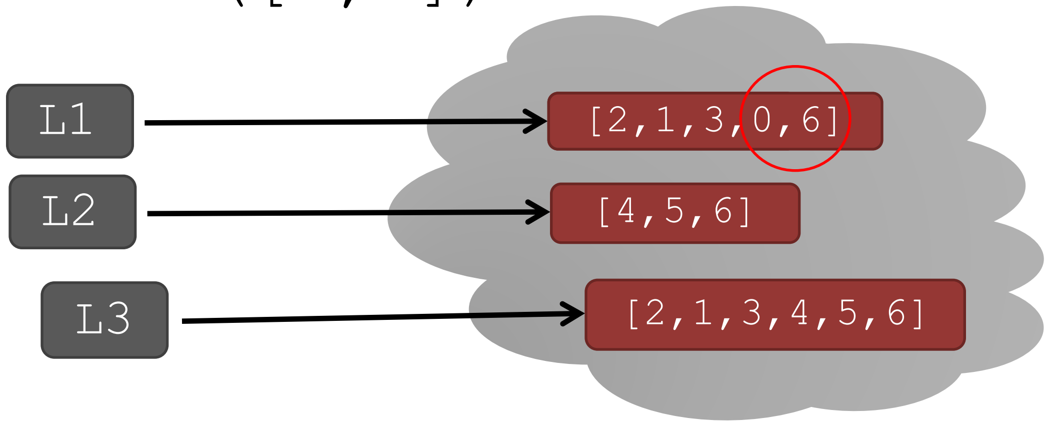
Remember strings

- **Concatenation**, + operator, creates a **new** list
- **Mutate** list with `L.extend(some_list)`

```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2
L1.extend([0,6])
```

→ `L3` is `[2,1,3,4,5,6]`

→ mutated `L1` to `[2,1,3,0,6]`



Note that L3 does not change after we mutate L1, since it was created as a new list from the original L1

TRICKY EXAMPLE 3: combining

```
L = [1,2,3,4]
for e in L:
    L = L + L
    print(L)
```

Originally [1,2,3,4]

1st time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4]

2nd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

3rd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

4th time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

L is **bound to a new object** each iteration;
but looping of e walks down structure pointed to when called, so iterates only 4 times, over original [1,2,3,4]

OPERATION ON LISTS: REMOVE

- Delete element at a **specific index** with `del(L[index])`
- Remove element at **end of list** with `L.pop()`, returns the removed element
- Remove a **specific element** with `L.remove(element)`
  - Looks for the element and removes it
  - If element occurs multiple times, removes first occurrence
  - If element not in list, gives an error

all these operations **mutate** the list

```
L = [2,1,3,6,3,7,0] # do below in order
L.remove(2) → mutates L = [1,3,6,3,7,0]
L.remove(3) → mutates L = [1,6,3,7,0]
del(L[1])   → mutates L = [1,3,7,0]
L.pop()     → returns 0 and mutates L = [1,3,7]
```

MUTATION AND ITERATION

http://www.pythontutor.com/ to see step-by-step

- **Avoid** mutating a list as you are iterating over it

```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)
```

```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

```
L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
remove_dups(L1, L2)
```

Clone list first
Note that `L1_copy = L1` does NOT clone

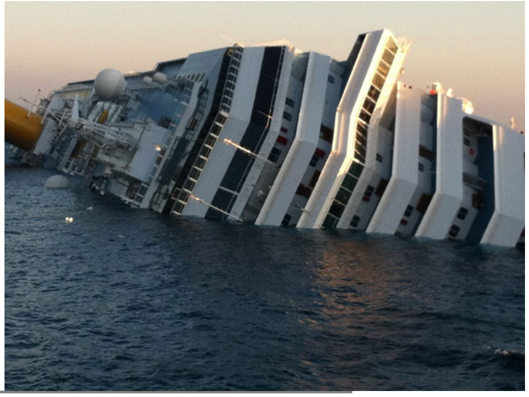
- `L1` is `[2,3,4]` not `[3,4]` Why?
  - Python uses an internal counter to keep track of index in the loop over list L1
  - Mutating changes the list but Python doesn't update the counter
  - Loop never sees element 2

CONVERT LISTS TO STRINGS AND BACK

- Convert **string to list** with `list(s)`, returns a list with every character from `s` an element in `L`
- Can use `s.split()`, to **split a string on a character** parameter, splits on spaces if called without a parameter
- Use `''.join(L)` to turn a **list of characters into a string**, can give a character in quotes to add char between every element

```
s = "I<3 cs"          → s is a string
list(s)               → returns ['I','<','3',' ','c','s']
s.split('<')          → returns ['I', '3 cs']
L = ['a','b','c']     → L is a list
''.join(L)            → returns "abc"
'_'.join(L)           → returns "a_b_c"
```

OTHER LIST OPERATIONS



- `sort()` and `sorted()`
- `reverse()`
- and many more!

https://docs.python.org/3/tutorial/datastructures.html

`L=[9,6,0,3]`

`a = sorted(L)` → returns sorted list, does **not mutate** `L`

`a = L.sort()` → **mutates** `L=[0,3,6,9]`, returns None

`L.reverse()` → **mutates** `L=[9,6,3,0]`

YOUR TURN


```
L1 = [‘re’]
L2 = [‘mi’]
L3 = [‘do’]
L4 = L1 + L2
L3.extend(l4)
L3.sort()
del(L3[0])
L3.append([‘fa’,’la’])
```

What is the value of L3 after you execute all the operations in this code?

A) `[‘mi’, ‘re’,[‘fa’, ‘la’]]`

B) `[‘mi’, ‘re’, ‘fa’, ‘la’]`

C) `[‘re’, ‘mi’ [‘fa’, ‘la’]]`

D) `[‘do’, ‘mi’, [‘fa’, ‘la’]]`

MUTATION, ALIASING, CLONING

IMPORTANT
and
TRICKY!

Again, Python Tutor is your best friend to help sort this out!

http://www.pythontutor.com/

LISTS IN MEMORY

- Lists are **mutable**
- Behave differently than immutable types
- A list is an object in memory
- Variable name points to object
- Using equal sign between mutable objects creates aliases
  - Both variables point to the same object in memory
- Any variable pointing to that object is affected by mutation of object
- Key phrase to keep in mind when working with lists is **side effects**

ALIASING

- City may be known by many names

- Attributes of a city
  - small, tech-savvy

Boston
The Hub
Beantown

- All nicknames point to the **same city**
  - add new attribute to **one nickname** …

Boston	small	tech-savvy	snowy

… all the **aliases** refer to the old attribute and all the new ones

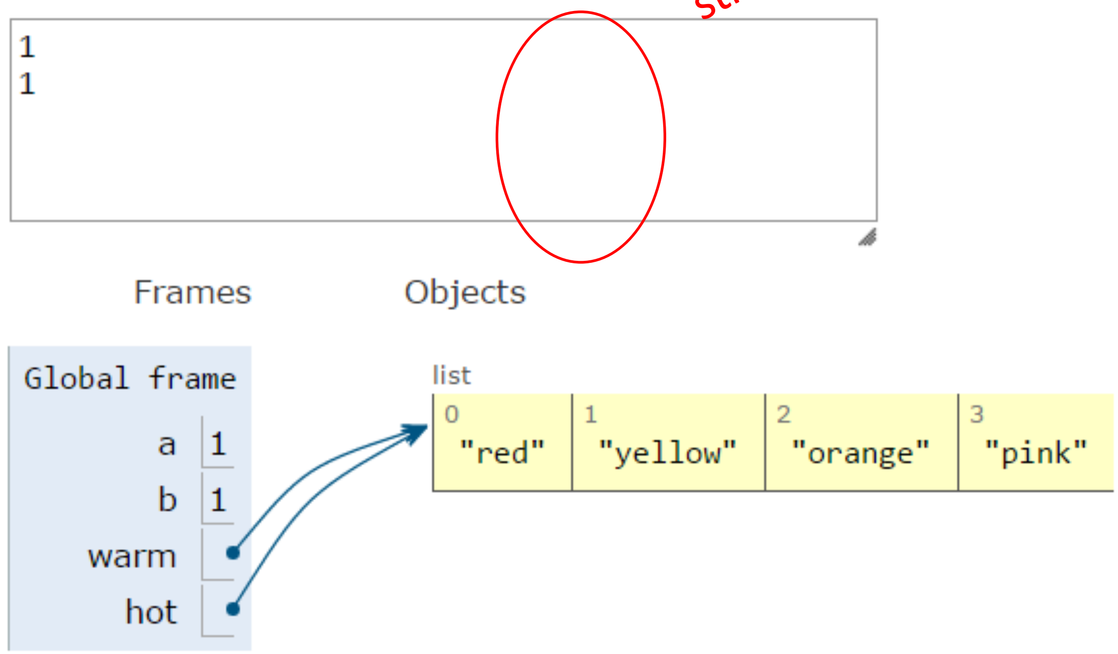
The Hub	small	tech-savvy	snowy
Beantown	small	tech-savvy	snowy

ALIASES

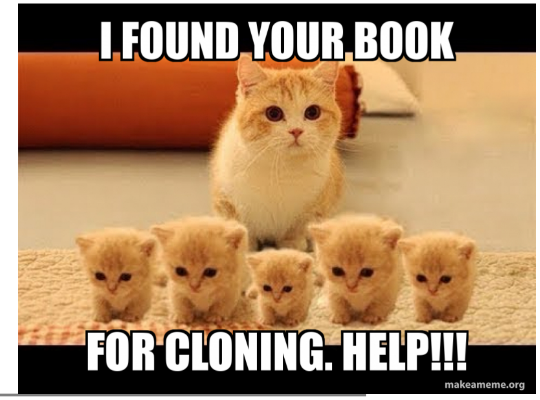
- `hot` is an **alias** for `warm` – changing one changes the other!
- `append()` has a side effect

Never explicitly changed warm, but structure has changed

```
a = 1
b = a
print(a)
print(b)

warm = ['red', 'yellow', 'orange']
hot = warm
```



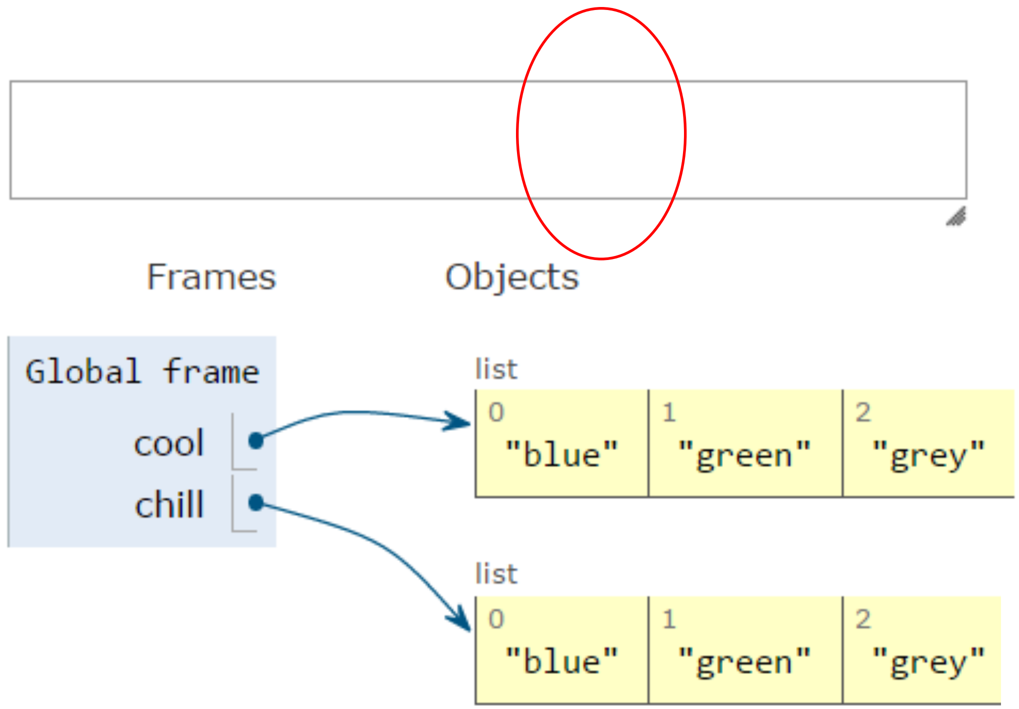
CLONING A LIST



- Create a new list and **copy every element** using a clone

```
chill = cool[:]
```

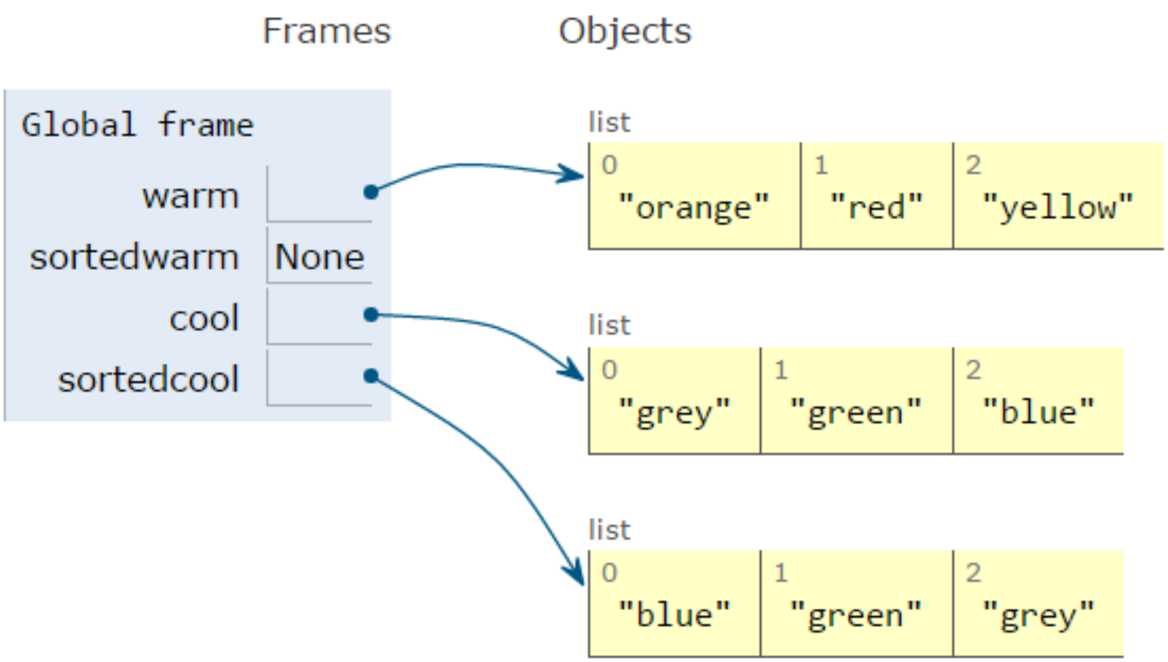
```
cool = ['blue', 'green', 'grey']
```



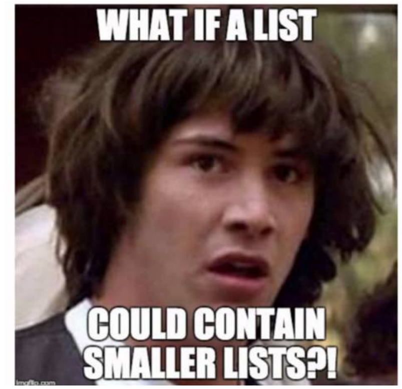
SORTING LISTS

- Calling `sort()` **mutates** the list, returns None
- Calling `sorted()` **does not mutate** list, must assign result to a variable

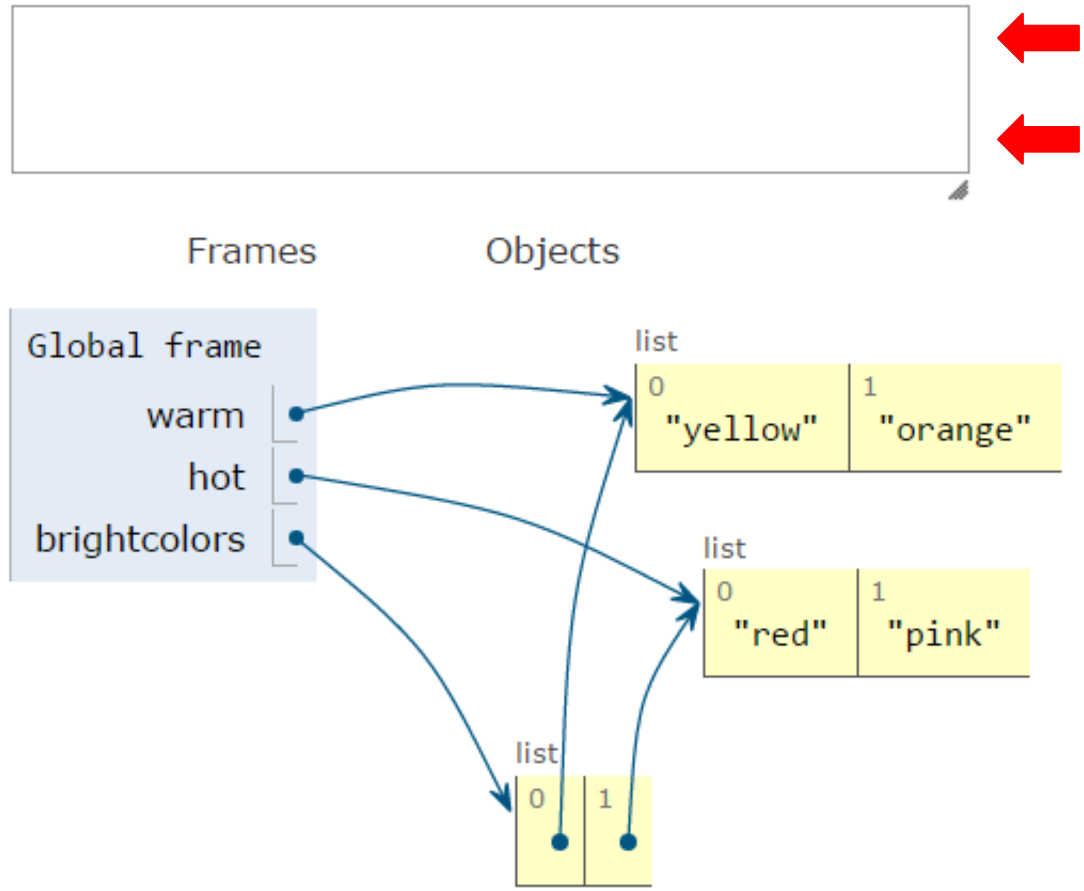
```
warm = ['red', 'yellow', 'orange']
```



LISTS OF LISTS OF LISTS OF....



- Can have **nested** lists
- Side effects still possible after mutation




```
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
```

YOUR TURN


```
L1 = ["bacon", "eggs"]
L2 = ["toast", "jam"]
brunch = L1
L1.append("juice")
brunch.extend(L2)
```

What is the value of brunch after you execute all the operations in this code?

A) `["bacon", "eggs", "toast", "jam"]`

B) `["bacon", "eggs", "juice", "toast", "jam"]`

C) `["bacon", "eggs", "juice", ["toast", "jam"]]`

D) `["bacon", "eggs", ["toast", "jam"]]`

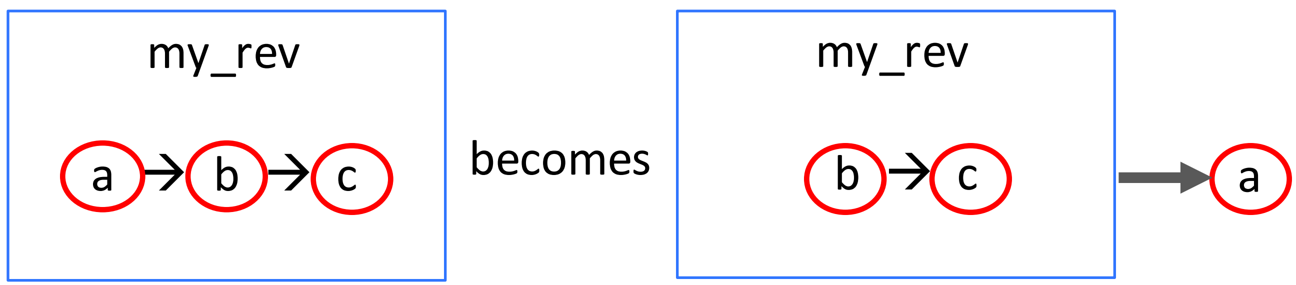
LISTS ARE NATURALLY RECURSIVE


```python
def total_iter(L):
    result = 0
    for e in L:
        result += len(e)
    return result


test = ["abc",'d',"efghi"]
print(total_iter(test))
```

```python
def total_recur(L):
    if L == []:
        return 0
    else:
        return len(L[0]) + \
               total_recur(L[1:])


test = ["abc",'d',"efghi"]
print(total_recur(test))
```

LISTS ARE NATURALLY RECURSIVE

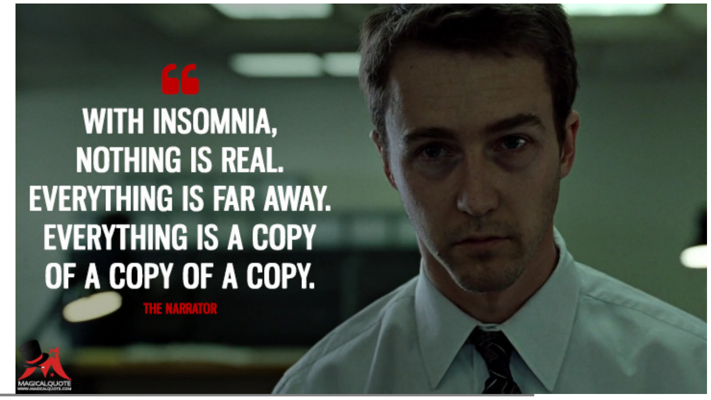
- The list operation `reverse` is built in; but we can easily see how a list naturally supports recursion
  - To reverse a list (as a copy), recursively reverse all but the first element, and add that element to the end




```
def my_rev(L):
    if L == []:
        return L
    else:
        return my_rev(L[1:]) + [L[0]]

test2 = ["abc", ['d'], ['e', ['f', 'g']]]
print(my_rev(test2))
```

Note: concatenation expects a pair of lists

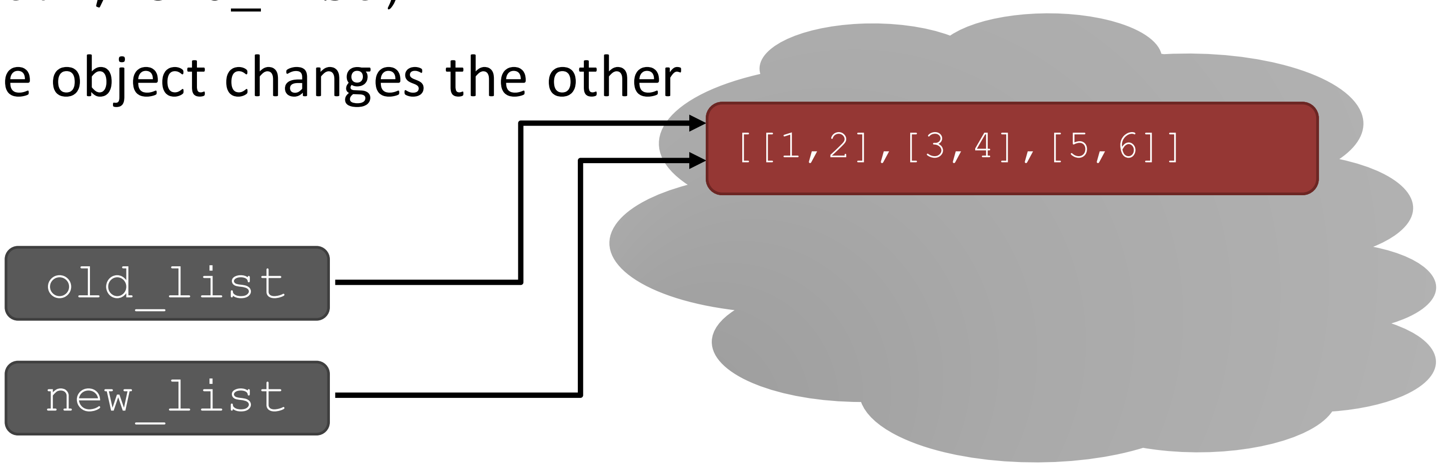
CONTROL COPYING



- Assignment just creates a new pointer to same object

```
old_list = [[1,2],[3,4],[5,'foo']]
new_list = old_list


new_list[2][1] = 6
print("New list:", new_list)
print("Old list:", old_list)
```

- So mutating one object changes the other



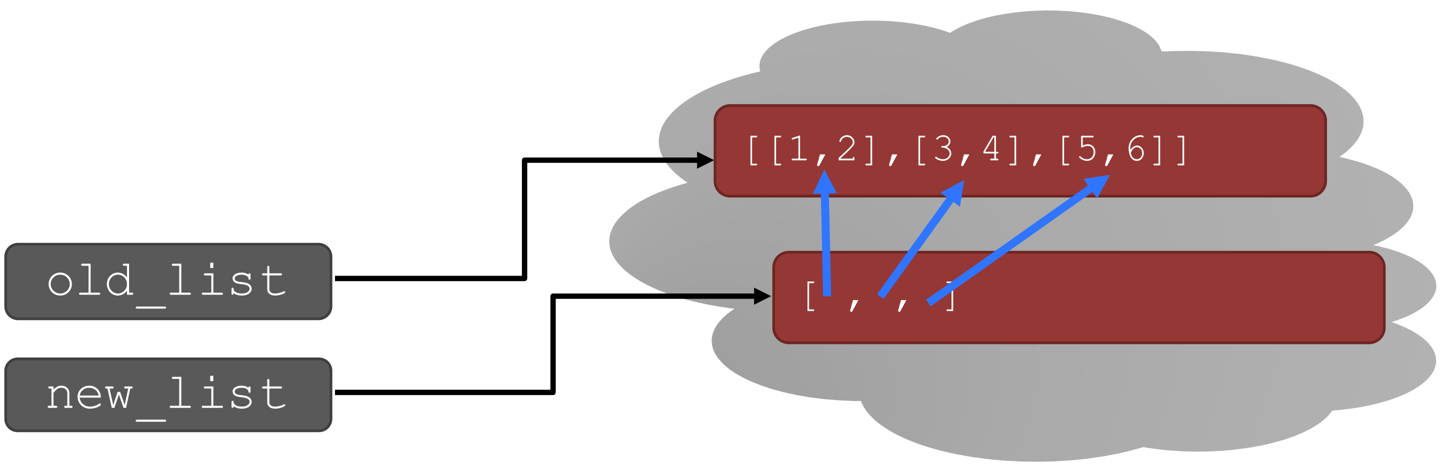
 6.0001 LECTURE 5

CONTROL COPYING

- Suppose we want to create a copy of a list, not just a shared pointer; shallow copying does this at the top level of the list

```
import copy
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)

print("New list:", new_list)
print("Old list:", old_list)
```

```
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)


print("New  list:", new_list)
print("Old  list:", old_list)
```

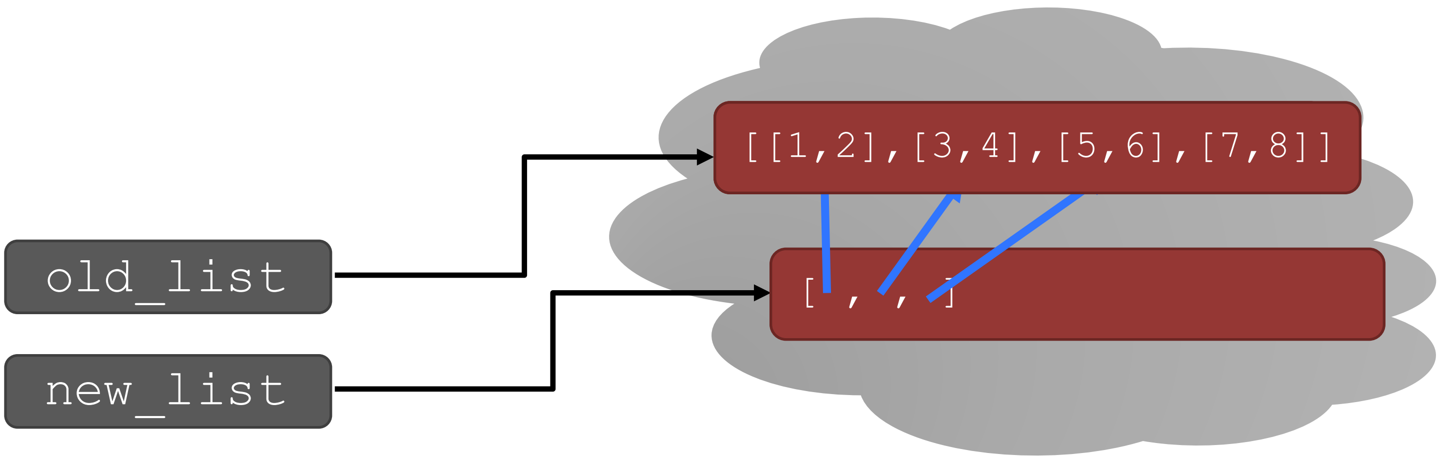


CONTROL COPYING

- Now we mutate the top level structure

```
import copy
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)

old_list.append([7,8])
print("New list:", new_list)
print("Old list:", old_list)
```

```
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)

old_list.append([7,8])
print("New list:", new_list)
print("Old list:", old_list)
```

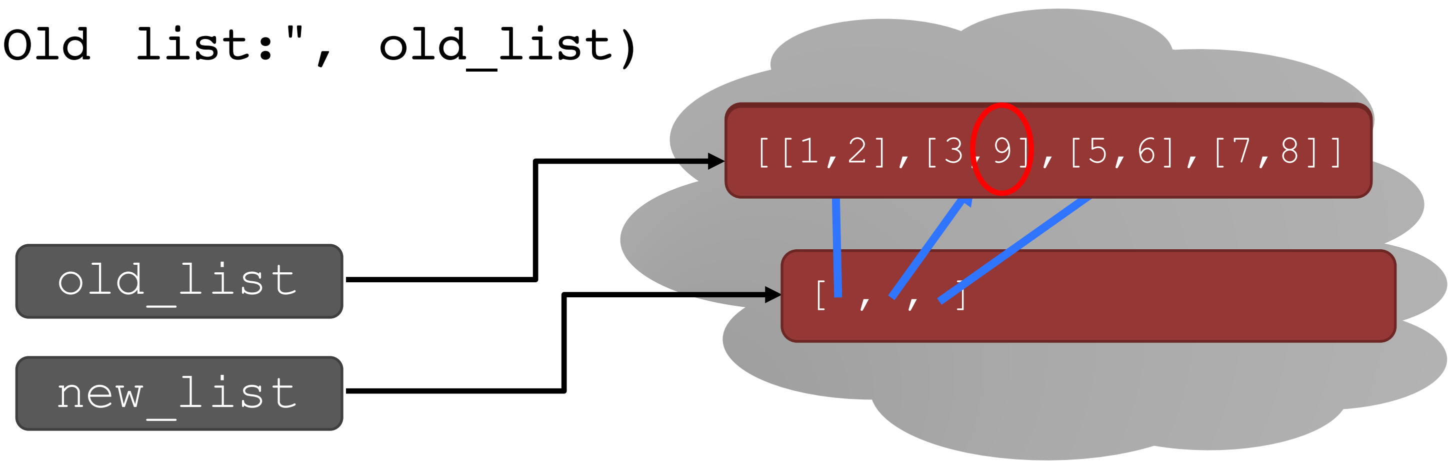


CONTROL COPYING

- But if we change an element in one of the sub-structures

```
import copy
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)

old_list.append([7,8])
old_list[1][1] = 9
print("New list:", new_list)
print("Old list:", old_list)
```

```
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)

old_list.append([7,8])
old_list[1][1] = 9
print("New list:", new_list)
print("Old list:", old_list)
```

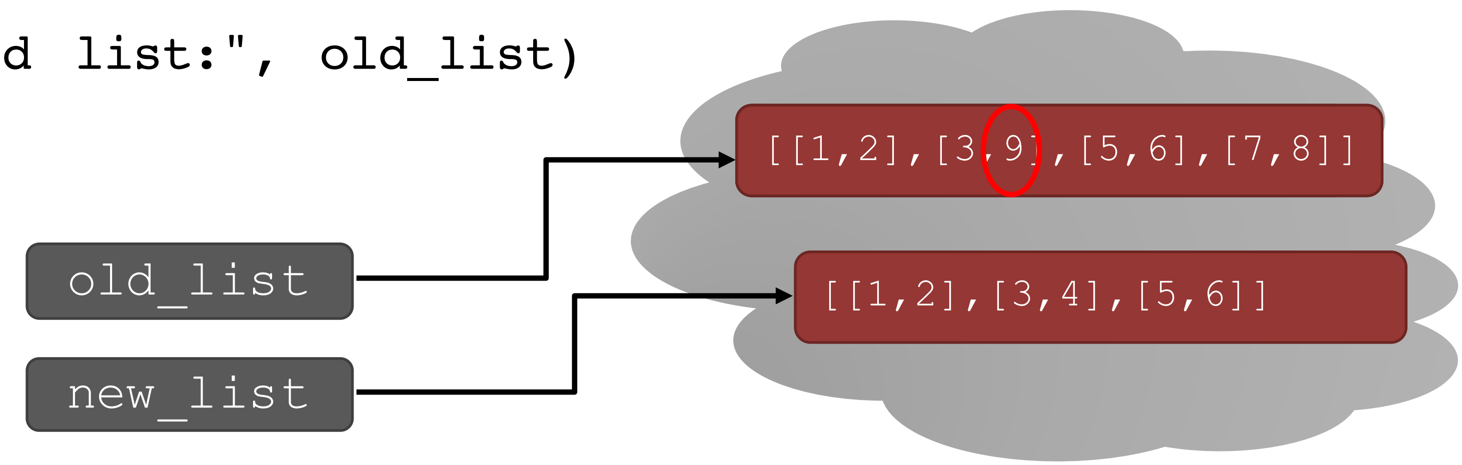


CONTROL COPYING

- If we want all structures to be new copies, we need a deep copy

```
import copy
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.deepcopy(old_list)


old_list.append([7,8])
old_list[1][1] = 9
print("New list:", new_list)
print("Old list:", old_list)
```

```
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.deepcopy(old_list)

old_list.append([7,8])
old_list[1][1] = 9
print("New list:", new_list)
print("Old list:", old_list)
```



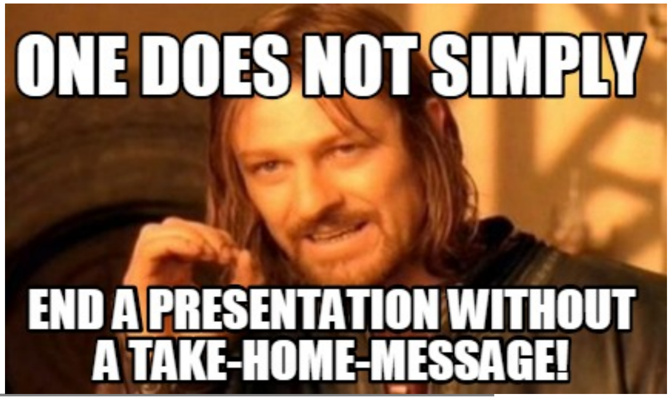
WRITING YOUR OWN VERSION


```
def my_deep_copy(L):
    if L == []:
        return L
    elif type(L[0]) == type([]):
        return [my_deep_copy(L[0])] +\
               my_deep_copy(L[1:])
    else:
        return [L[0]] + my_deep_copy(L[1:])
```

WHY LISTS AND TUPLES?

- If mutation can cause so many problems, why do we even want to have lists, why not just use tuples?
  - Efficiency – if processing very large sequences, don't want to have to copy every time we change an element
- If lists basically do everything that tuples do, why not just have lists?
  - Immutable structures can be very valuable, e.g., we will see using immutable structures as keys into dictionaries next lecture

Take home message



- Lists and tuples provide compound data structures
  - Can be indexed
  - Can be sliced
- They naturally support recursive or iterative algorithms
- Many built in methods for processing lists – reverse, sort, sorted, append, extend, etc.
- Lists are mutable!
  - Need to be careful about aliasing – when two names refer to the same mutable structure