ALIASING, CLONING

(download slides and .py files to follow along)

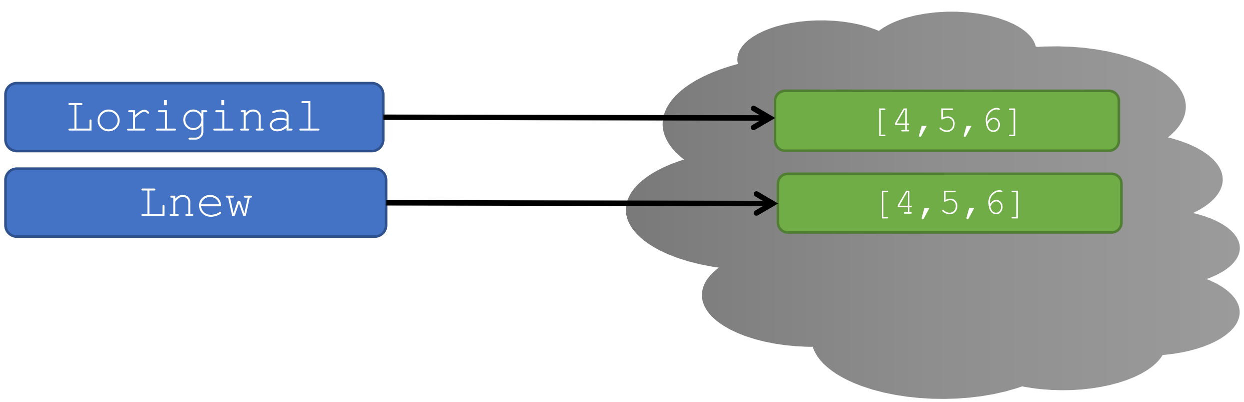
6.100L Lecture 11

Ana Bell

MAKING A COPY OF THE LIST

- Can **make a copy of a list object** by duplicating all elements (top-level) into a new list object
- `Lcopy = L[:]`
  - Equivalent to looping over L and appending each element to Lcopy
  - This does not make a copy of elements that are lists (will see how to do this at the end of this lecture)

```
Loriginal = [4,5,6]
Lnew = Loriginal[:]
```



YOU TRY IT!

- Write a function that meets the specification.
- Hint. Make a copy to save the elements. The use L.clear() to empty out the list and repopulate it with the ones you're keeping.

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None
    """


L = [1,2,2,2]
remove_all(L, 2)
print(L)     # prints [1]
```

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 (can also call with specific index: `L.pop(3)`)
- Remove a **specific element** with `L.remove(element)`
  - Looks for the element and removes it (mutating the list)
  - 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]
a = L.pop()     → returns 0 and mutates L = [1,3,7]
```

EXERCISE WITH REMOVE INSTEAD OF COPY AND CLEAR

- Rewrite the code to remove `e` as long as we still had it in the list
- It works well!

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    while e in L:
        L.remove(e)
```

EXERCISE WITH REMOVE INSTEAD OF COPY AND CLEAR

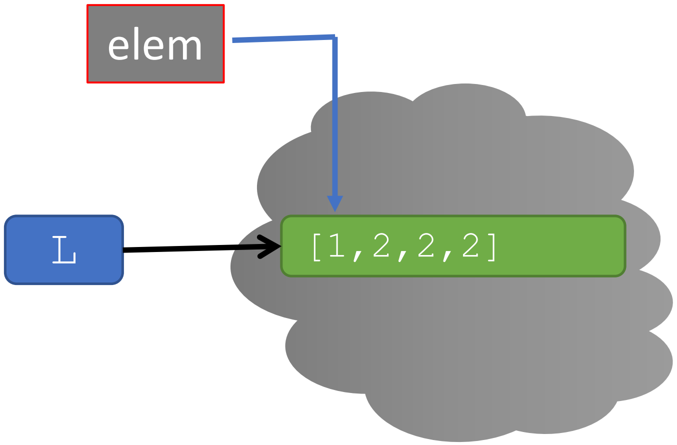
- What if the code was this:

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    for elem in L:
        if elem == e:
            L.remove(e)

L = [1,2,2,2]
remove_all(L, 2)
print(L)      # should print [1]
```

Actually prints [1,2]

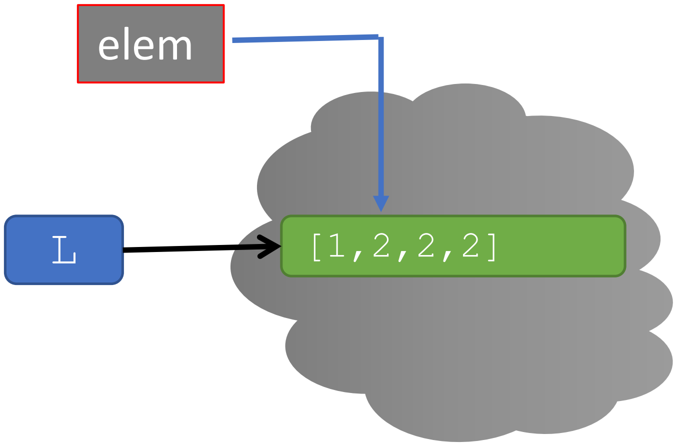
EXERCISE WITH REMOVE INSTEAD OF COPY AND CLEAR

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    for elem in L:
        if elem == e:
            L.remove(e)

L = [1,2,2,2]
remove_all(L, 2)
print(L)     # should print [1]
```



EXERCISE WITH REMOVE INSTEAD OF COPY AND CLEAR

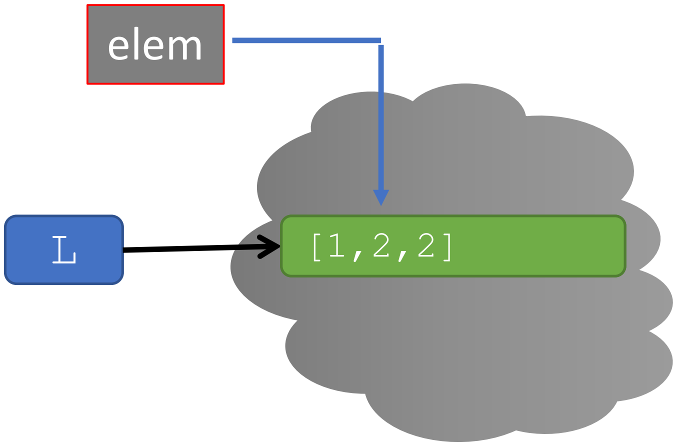
```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    for elem in L:
        if elem == e:
            L.remove(e)

L = [1,2,2,2]
remove_all(L, 2)
print(L)    # should print [1]
```



EXERCISE WITH REMOVE INSTEAD OF COPY AND CLEAR

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    for elem in L:
        if elem == e:
            L.remove(e)

L = [1,2,2,2]
remove_all(L, 2)
print(L)    # should print [1]
```

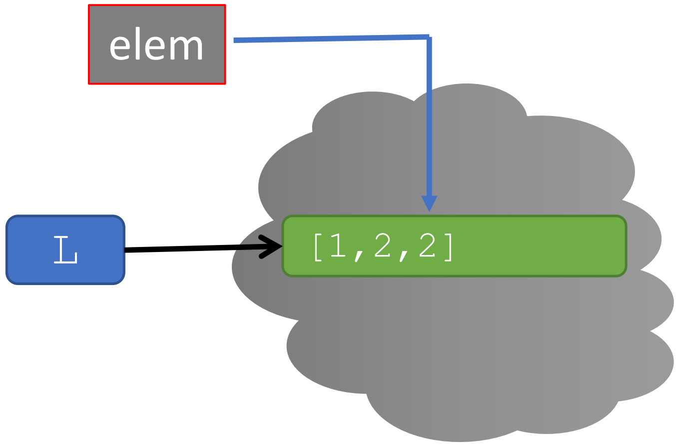
Removes the 2, but doesn't shift the pointer back!



EXERCISE WITH REMOVE INSTEAD OF COPY AND CLEAR

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    for elem in L:
        if elem == e:
            L.remove(e)

L = [1,2,2,2]
remove_all(L, 2)
print(L)     # should print [1]
```

elem moves forward in the sequence

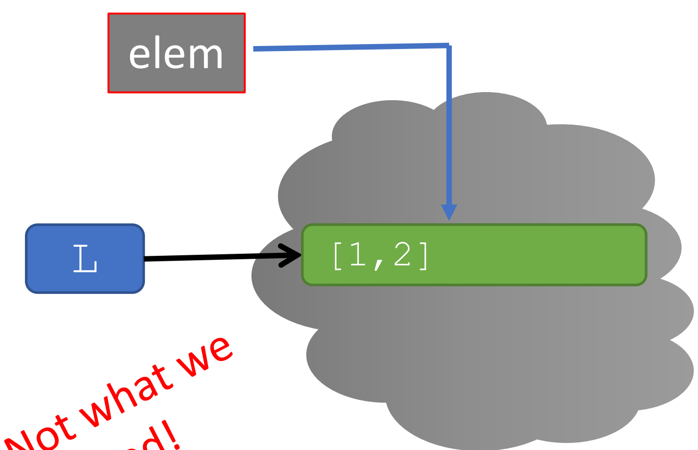


EXERCISE WITH REMOVE INSTEAD OF COPY AND CLEAR

- It's not correct! We **removed items as we iterated over the list**!

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    for elem in L:
        if elem == e:
            L.remove(e)

L = [1,2,2,2]
remove_all(L, 2)
print(L)    # should print [1]
```

Remove the 2, and done



Not what we wanted!

TRICKY EXAMPLES OVERVIEW

- TRICKY EXAMPLE 1:
  - A loop iterates over **indices of L** and **mutates L** each time (adds more elements).
- TRICKY EXAMPLE 2:
  - A loop iterates over **L's elements** directly and **mutates L** each time (adds more elements).
- TRICKY EXAMPLE 3:
  - A loop iterates over **L's elements** directly but **reassigns L** to a new object each time
- TRICKY EXAMPLE 4:
  - A loop iterates over **L's elements** directly and mutates L by **removing elements**.

TRICKY EXAMPLE 4

PYTHON TUTOR LINK to see step-by-step

- Want to mutate L1 to remove any elements that are also in L2

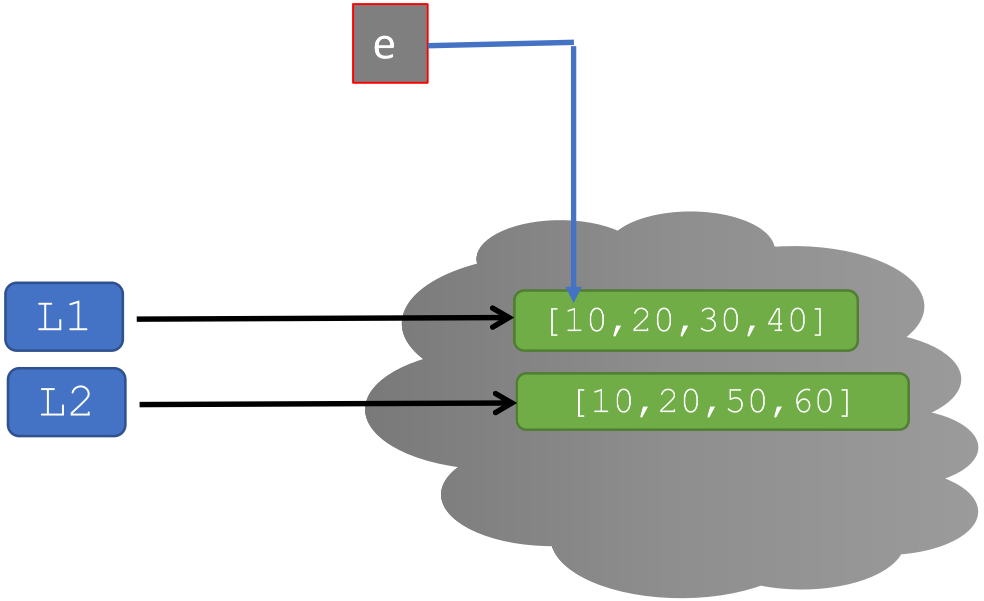
```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```

Want a function that returns a list, where every element also in a second list is removed.

- `L1` is `[20,30,40]` not `[30,40]` Why?
  - You are **mutating a list as you are iterating over it**
  - Python uses an internal counter. Tracks of index in the loop over list L1
  - Mutating changes the list but Python doesn't update the counter
  - Loop never sees element 20

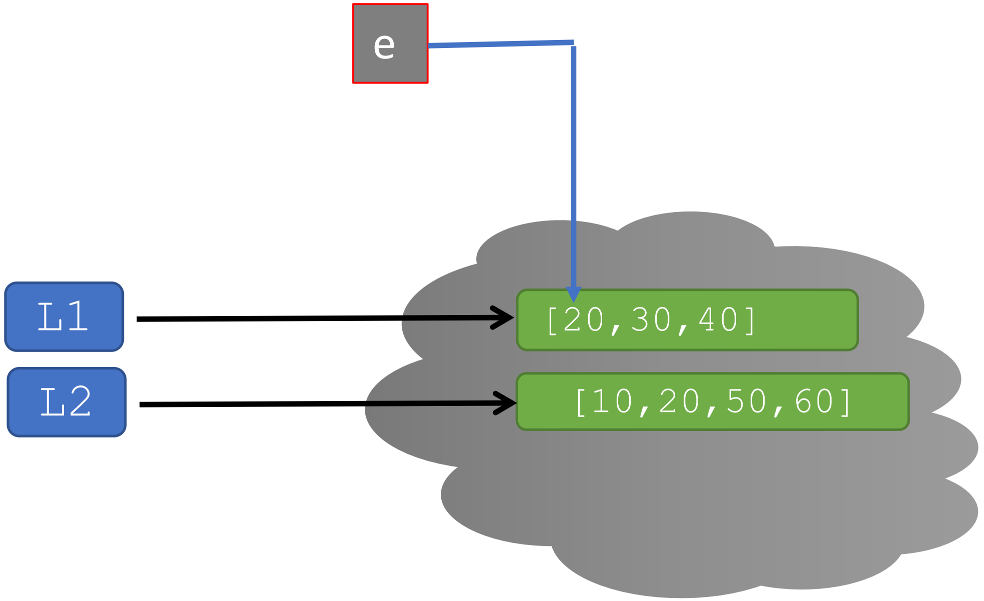
MUTATION AND ITERATION WITHOUT CLONE

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

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



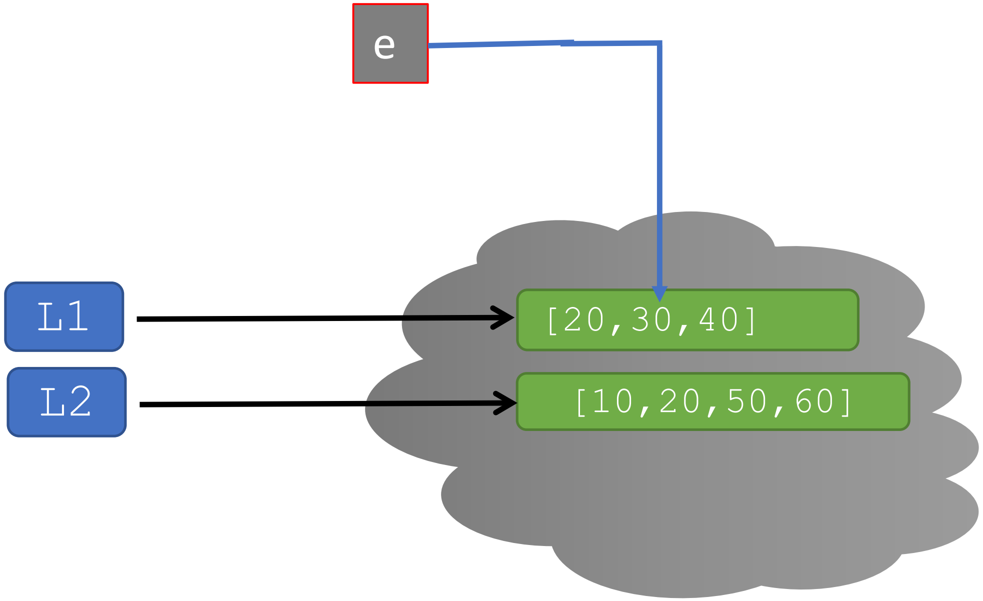
MUTATION AND ITERATION WITHOUT CLONE

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

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



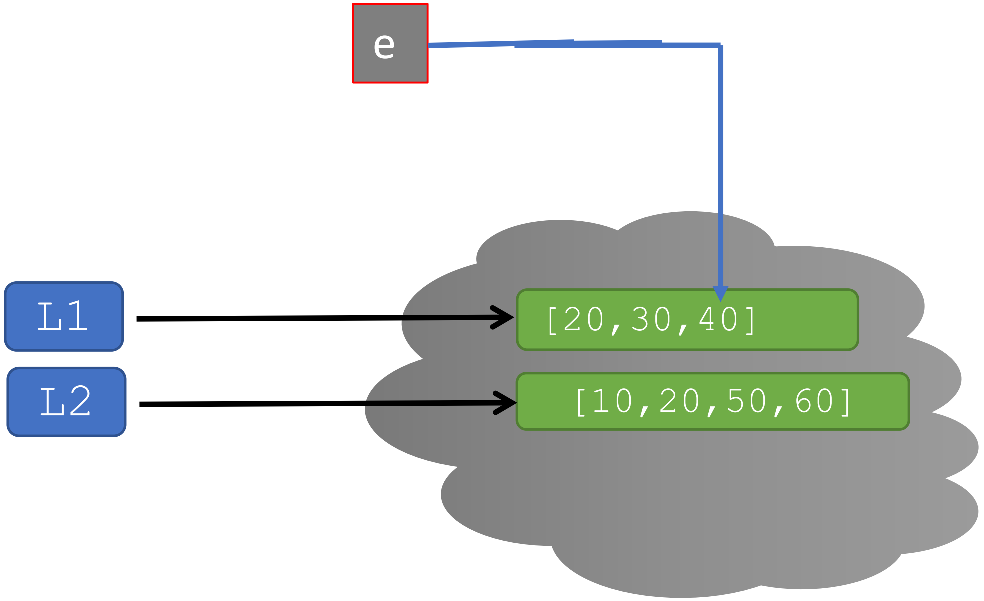
MUTATION AND ITERATION WITHOUT CLONE

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

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



MUTATION AND ITERATION WITHOUT CLONE

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

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



MUTATION AND ITERATION **WITH CLONE**

`L1_copy = L1[:]`

- Make a **clone** with [:]

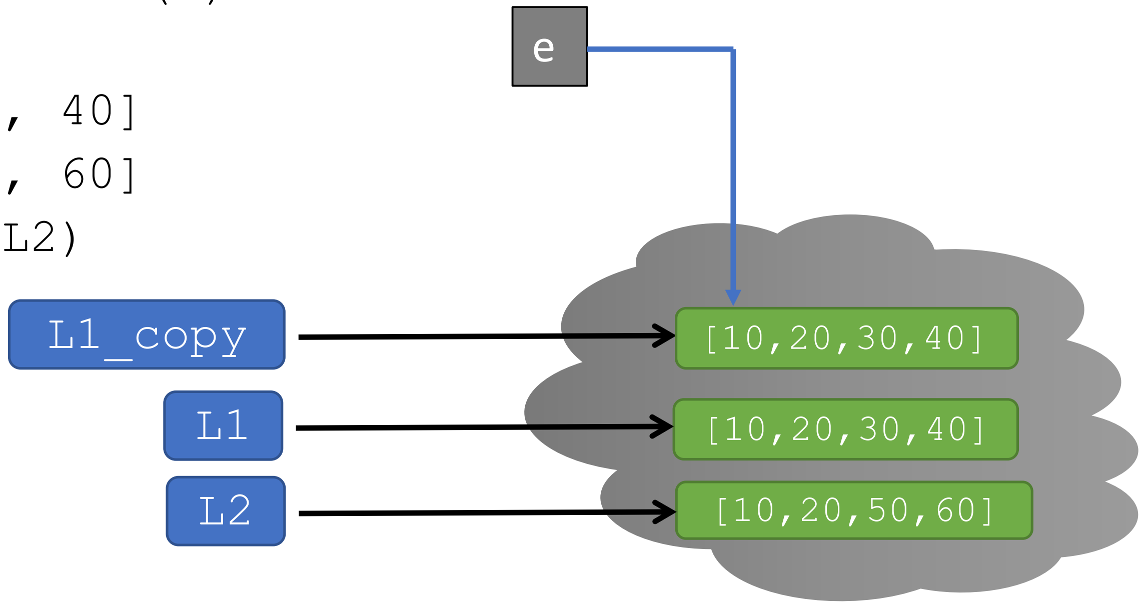
```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)
```

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

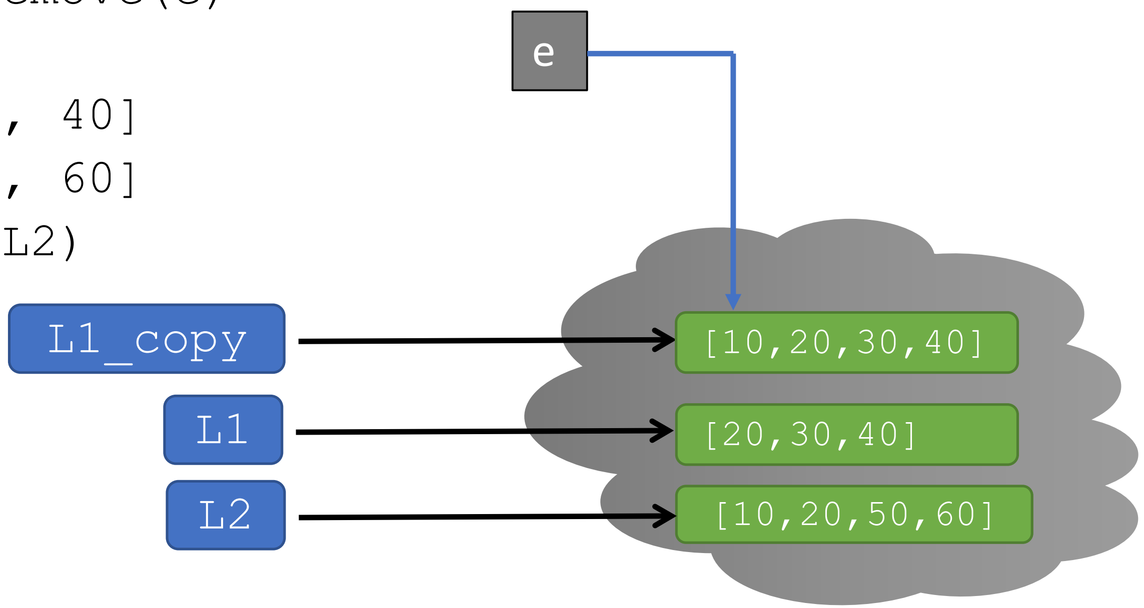
```
L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```

- New version works!
  - Iterate over a copy
  - Mutate original list, not the copy
  - Indexing is now consistent

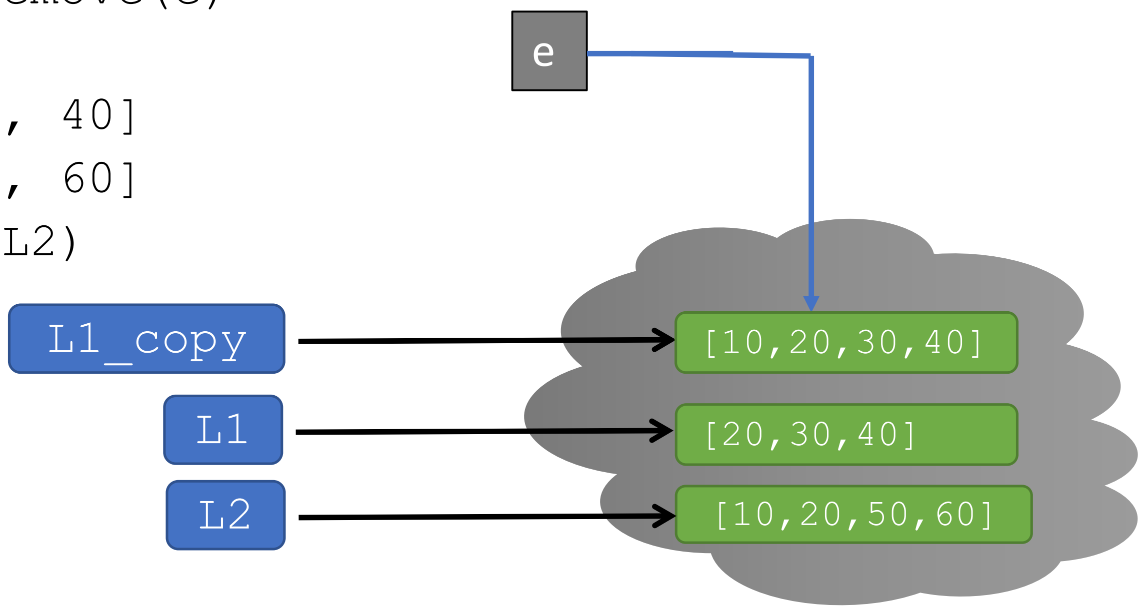
```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



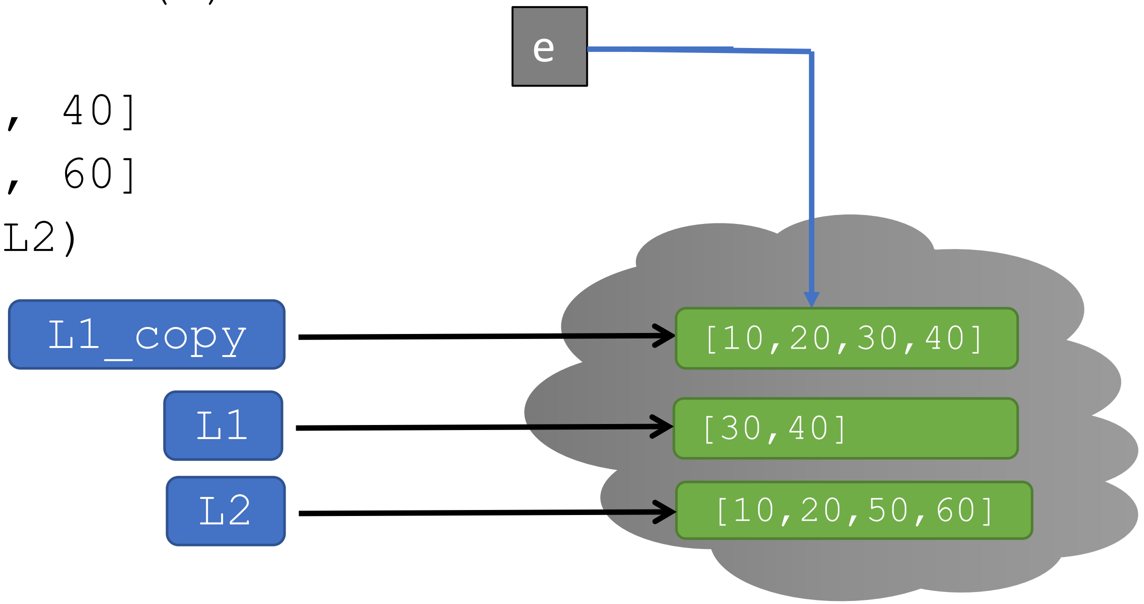
```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



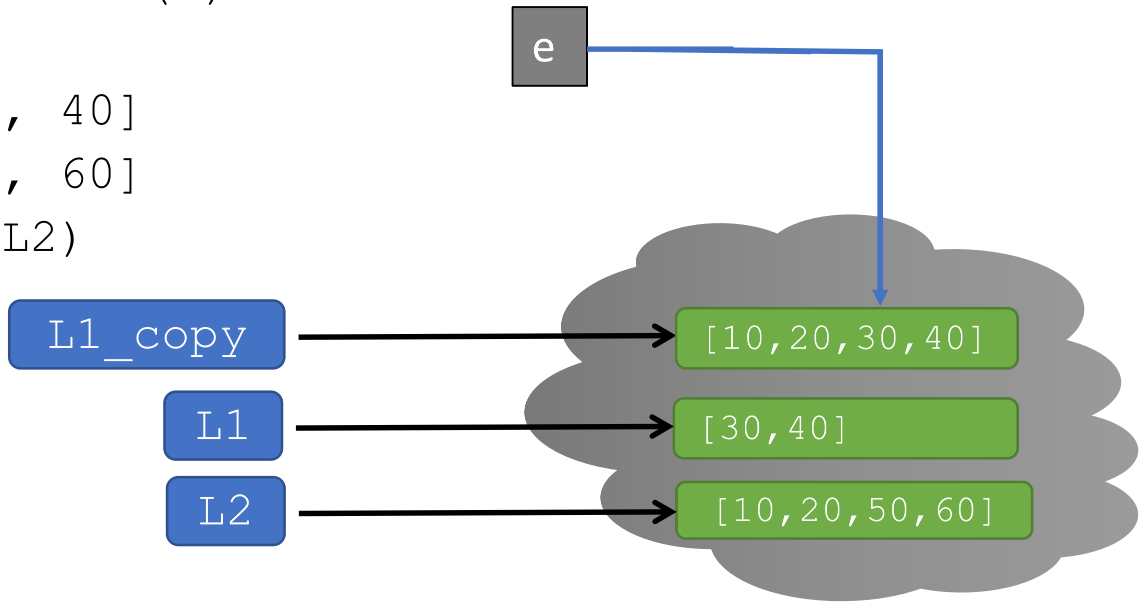
```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



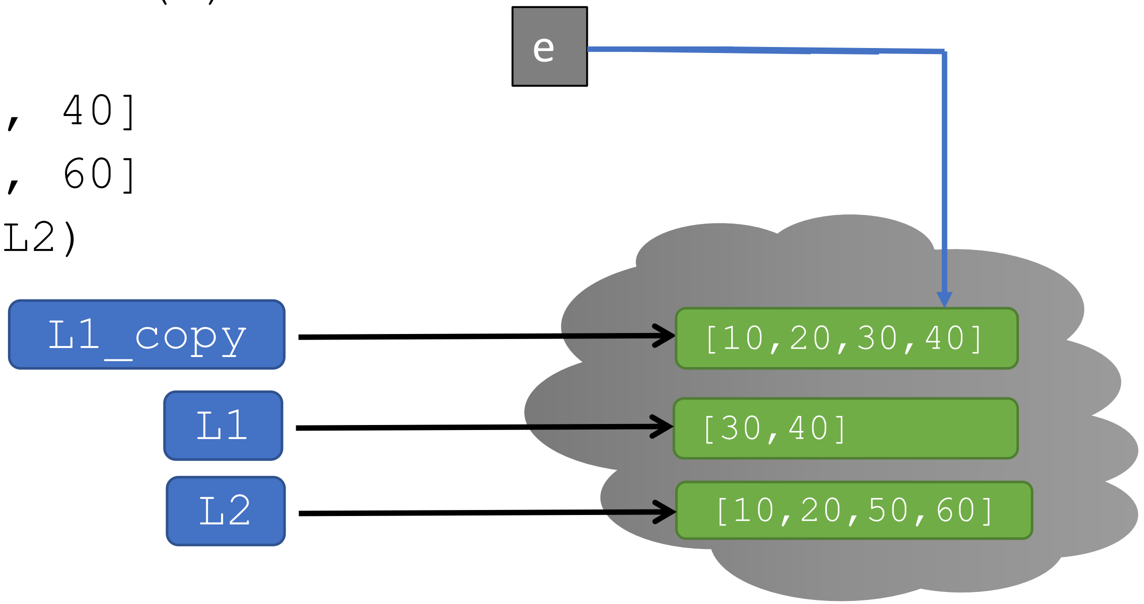
```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



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

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



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

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



ALIASING

- City may be known by many names
- Attributes of a city
  - Small, tech-savvy
- All nicknames point to the **same city**
  - Add new attribute to **one nickname** …

Boston
The Hub
Beantown
Athens of America

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

MUTATION AND ITERATION **WITH ALIAS**

`L1_copy = L1`

- Assignment (= sign) on mutable obj creates an **alias**, not a clone

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

The same object as L1

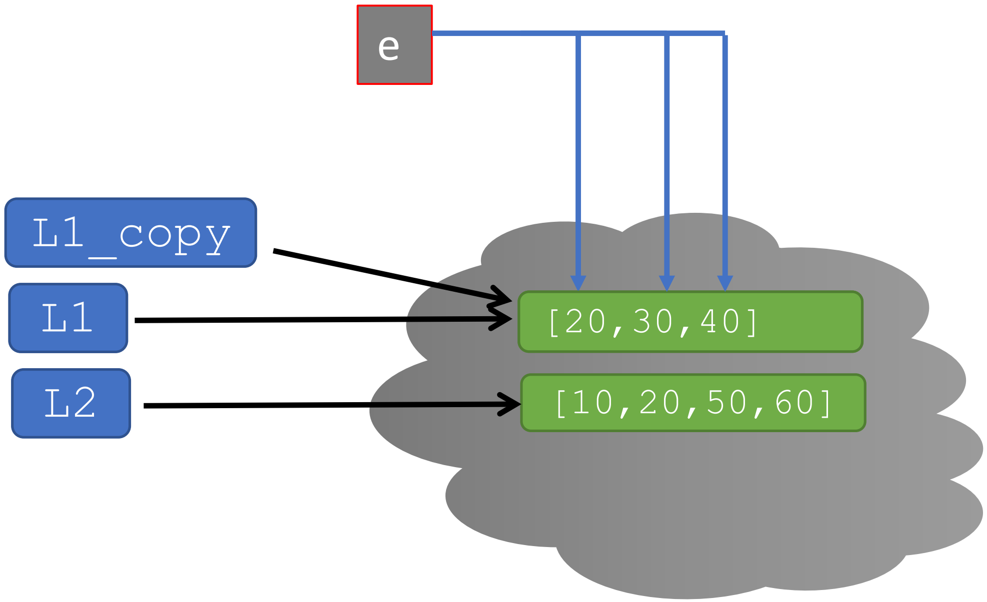
```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

```
L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```

Note that `L1_copy = L1` does NOT clone

- Using a simple assignment without making a copy
  - Makes an alias for list (**same list object referenced by another name**)
  - It's like iterating over L itself, it doesn't work!

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

L1 = [10, 20, 30, 40]
L2 = [10, 20, 50, 60]
remove_dups(L1, L2)
```



BIG IDEA

When you pass a list as a parameter to a function, you are making an alias.

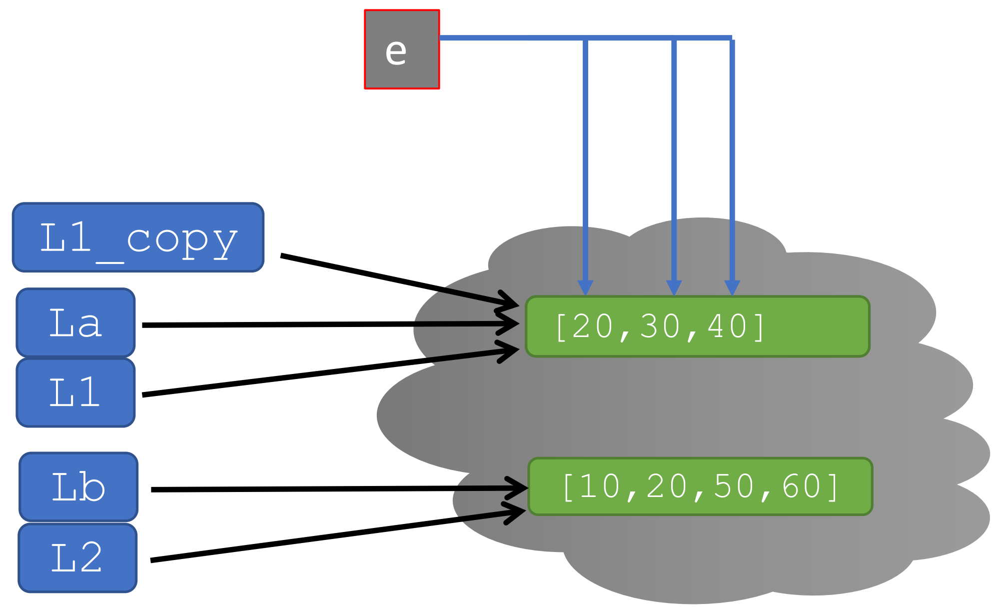
The **actual parameter** (from the function **call**) is an **alias** for the **formal parameter** (from the function **definition**).

Alias for La

Alias for Lb

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

La = [10, 20, 30, 40]
Lb = [10, 20, 50, 60]
remove_dups(La, Lb)
print(La)
```

L1 was mutated, but it's an alias for La



ALIASES, SHALLOW COPIES, AND DEEP COPIES WITH MUTABLE ELEMENTS

CONTROL COPYING

- Assignment just creates a new pointer to same object

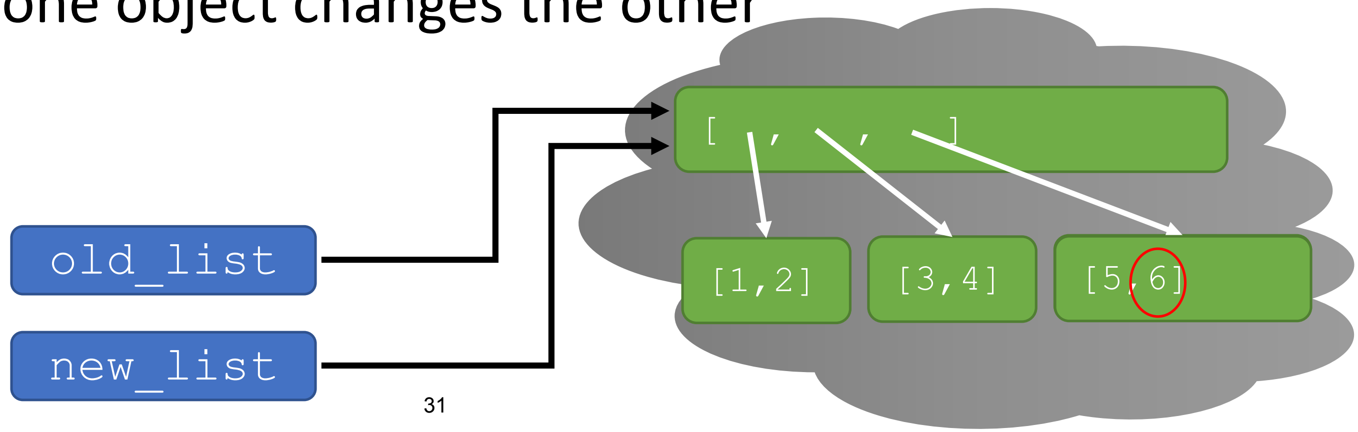
```
old_list = [[1,2],[3,4],[5,'foo']]
new_list = old_list
```

Assignment creates an alias for an existing data structure

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

```
New list: [[1,2],[3,4],[5,6]]
Old list: [[1,2],[3,4],[5,6]]
```

- So mutating one object changes the other



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**
    - Equivalent to syntax [:]
    - Any mutable elements are NOT copied
- Use this when your list contains immutable objects only

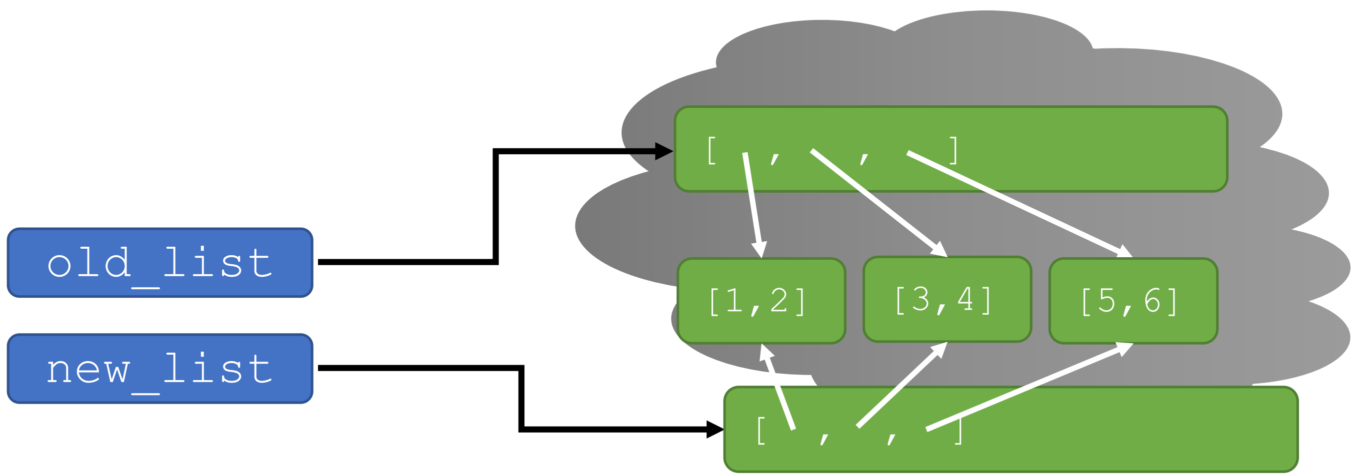
```
import copy
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)

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

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

Copy creates a new data structure, but actual elements are shared

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

```
New list: [[1,2],[3,4],[5,6]]
Old list: [[1,2],[3,4],[5,6]]
```



CONTROL COPYING

- Now we mutate the top level structure

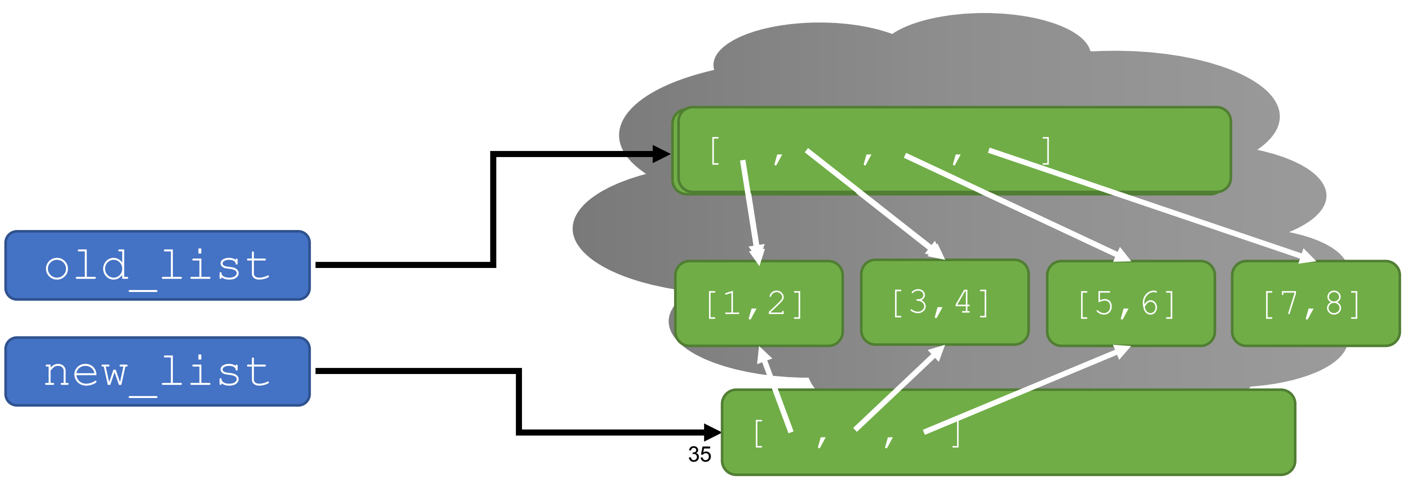
```
import copy
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)

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)
```

Copy creates a new data structure with shared elements, so mutating top level structure of one does not affect the clone

```
New list: [[1,2],[3,4],[5,6]]
Old list: [[1,2],[3,4],[5,6],[7,8]]
```



CONTROL COPYING

- But if we change an element in one of the sub-structures, they are shared!
- If your elements are not mutable then this is not a problem

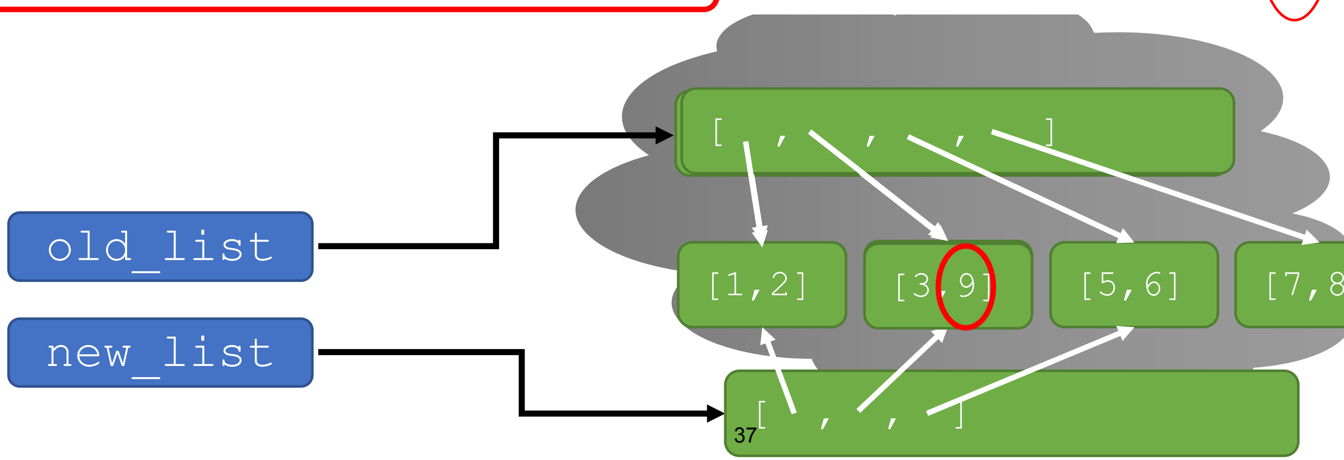
```
import copy
old_list = [[1,2],[3,4],[5,6]]
new_list = copy.copy(old_list)

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)
```

Shallow copying creates a new data structure with shared elements; so top level are clones, but elements are aliases; mutating an element will thus affect the other object

```
New list: [[1,2],[3,9],[5,6]]
Old list: [[1,2],[3,9],[5,6],[7,8]]
```



CONTROL COPYING

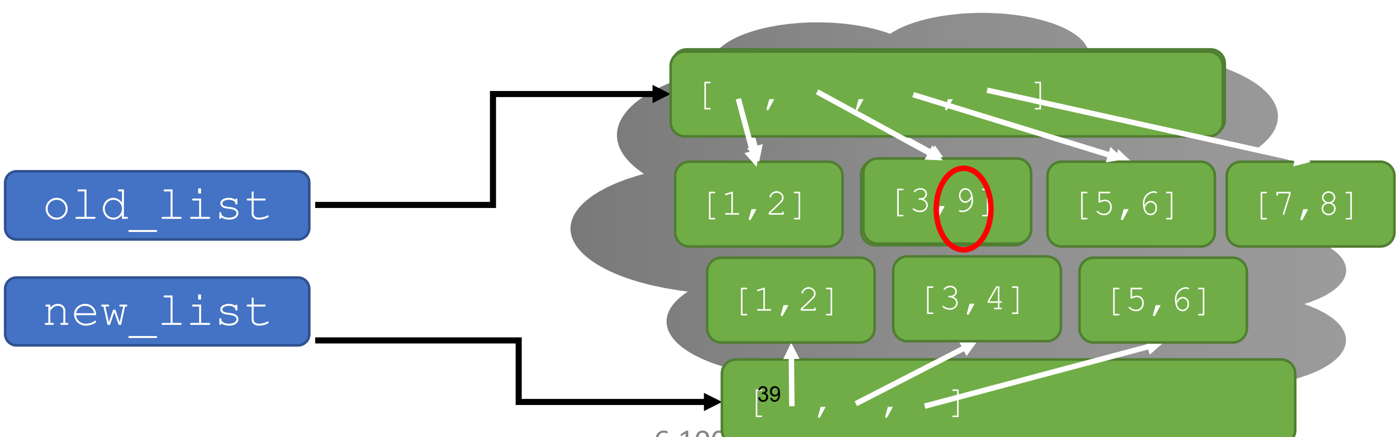
- If we want all structures to be new copies, we need a deep copy
- Use deep copy when your list might have mutable elements to ensure every structure at every level is copied

```
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)
```

```
New list: [[1,2],[3,4],[5,6]]
Old list: [[1,2],[3,9],[5,6],[7,8]]
```

Deep copying creates clones at all levels of structure; thus mutating one does not affect the other at any level



LISTS in MEMORY

- Separate the idea of the **object vs. the name we give** an object
  - A list is an object in memory
  - Variable name points to object
- Lists are **mutable** and behave differently than immutable types
- 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, even if mutation is by referencing another name
- If you want a copy, you explicitly tell Python to make a copy
- Key phrase to keep in mind when working with lists is **side effects,** especially when dealing with **aliases** – two names pointing to the same structure in memory
- ***Python Tutor is your best friend to help sort this out!*** http://www.pythontutor.com/

WHY LISTS and TUPLES?

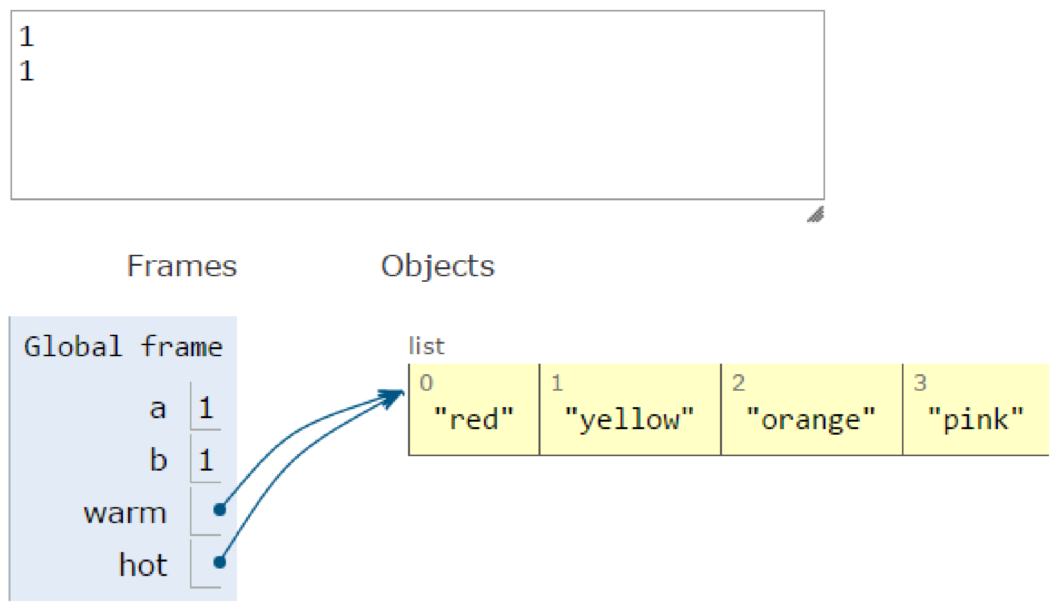
- If mutation can cause so many problems, why do we even want to have lists, **why not just use tuples**?
  - Efficiency – if processing very large sequences, don't want to have to copy every time we change an element
- If lists basically do everything that tuples do, **why not just have lists**?
  - Immutable structures can be very valuable in context of other object types
  - Don't want to accidentally have other code mutate some important data, tuples safeguard against this
  - They can be a bit faster

AT HOME TRACING EXAMPLES SHOWCASING ALIASING AND CLONING

ALIASES

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

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

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



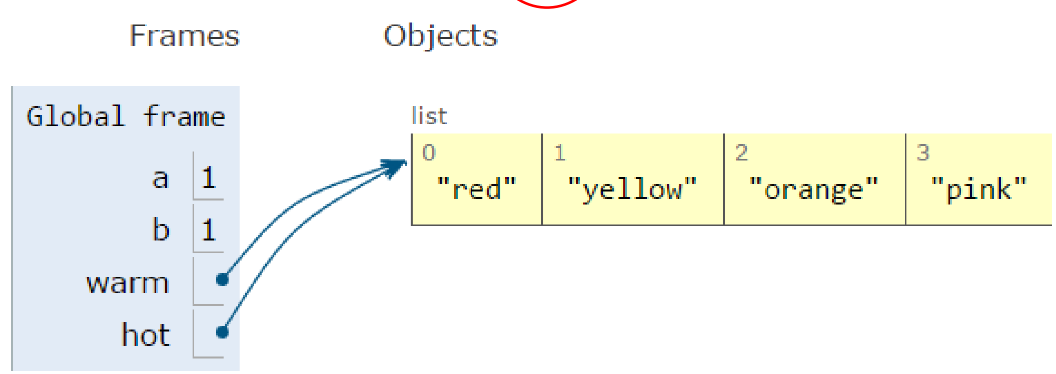
ALIASES

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

Never explicitly changed warm, but its value has changed

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

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

```
1
1
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```

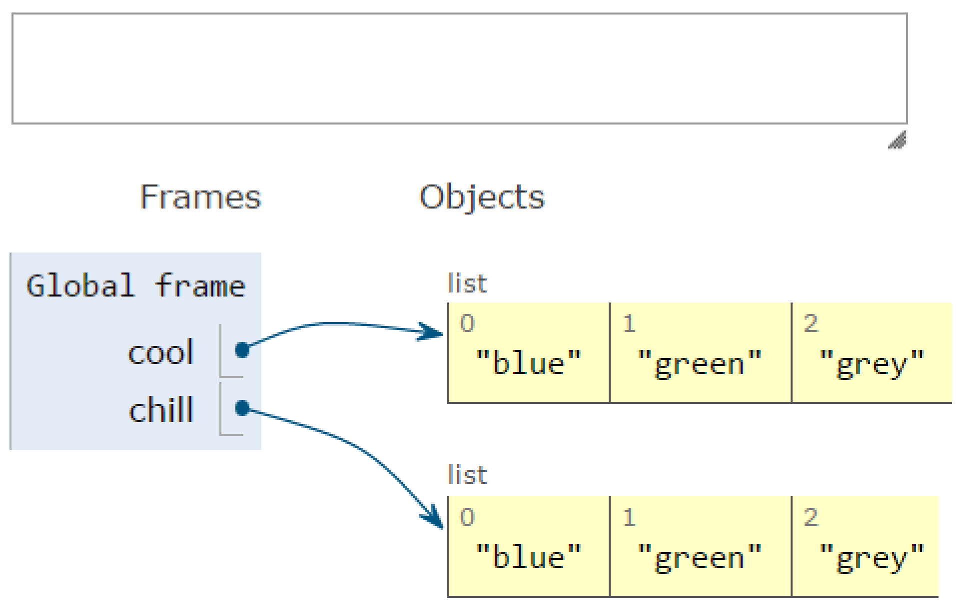


CLONING A LIST

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

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

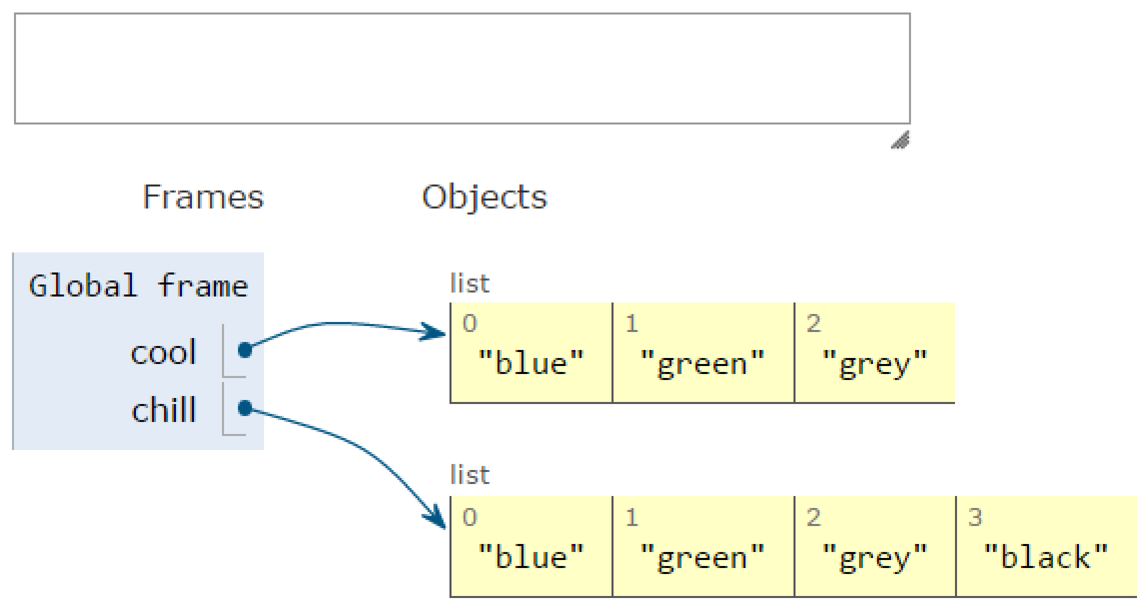
```
cool = ['blue', 'green', 'grey']
```



CLONING A LIST

- Create a new list and **copy every element** using a clone
  `chill = cool[:]`

```
cool = ['blue', 'green', 'grey']
chill = cool[:]
chill.append('black')
print(chill)
print(cool)
```



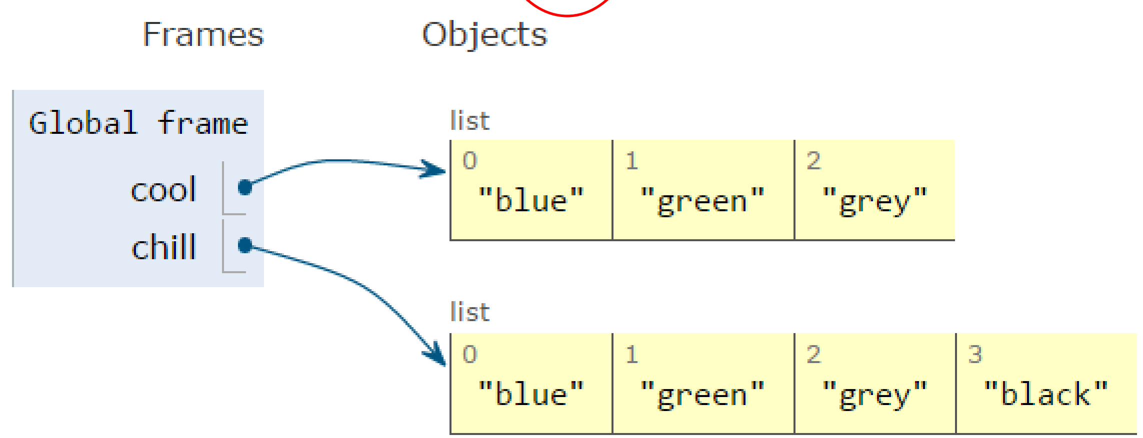
CLONING A LIST

- Create a new list and **copy every element** using a clone
  `chill = cool[:]`

```
cool = ['blue', 'green', 'grey']
chill = cool[:]
chill.append('black')
print(chill)
print(cool)
```

```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```

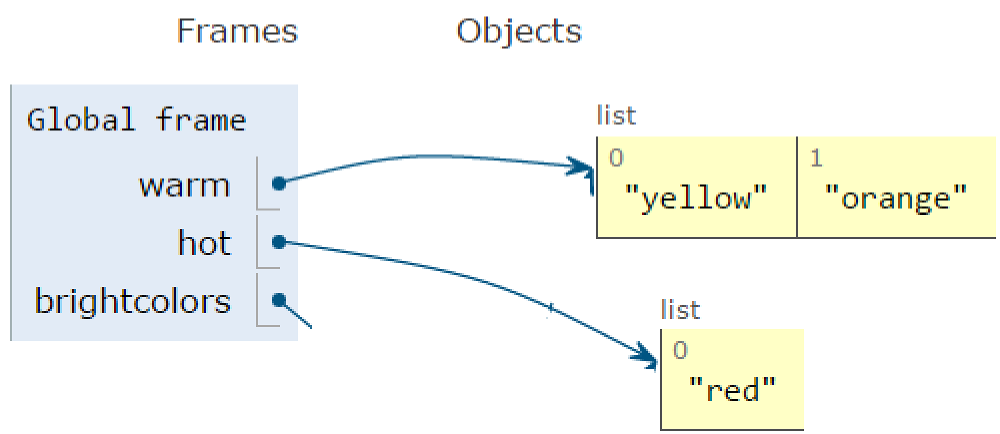
Because chill is a clone, changing chill does not change cool



LISTS OF LISTS OF LISTS OF….

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

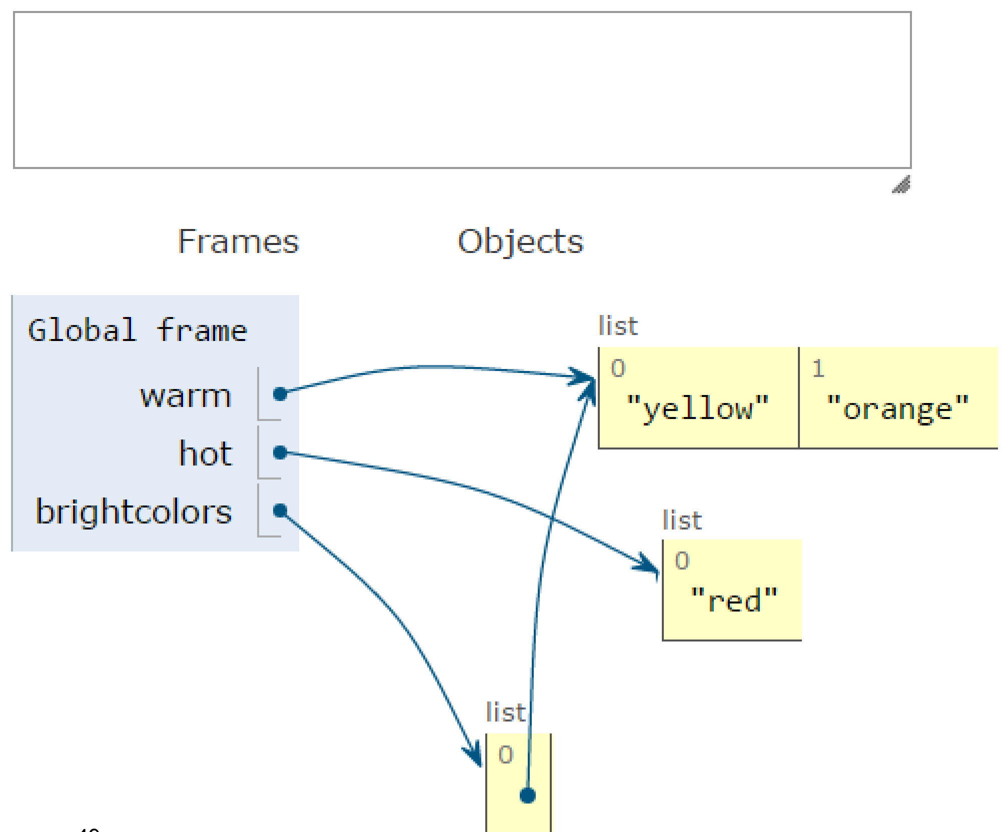
```
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
```



LISTS OF LISTS OF LISTS OF....

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

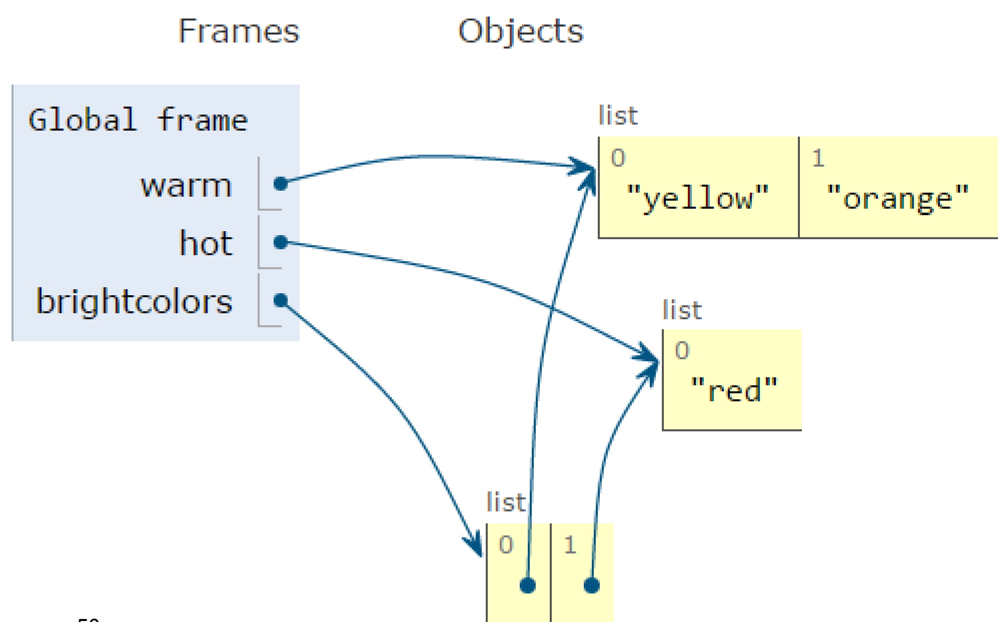
```
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
```



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)
```

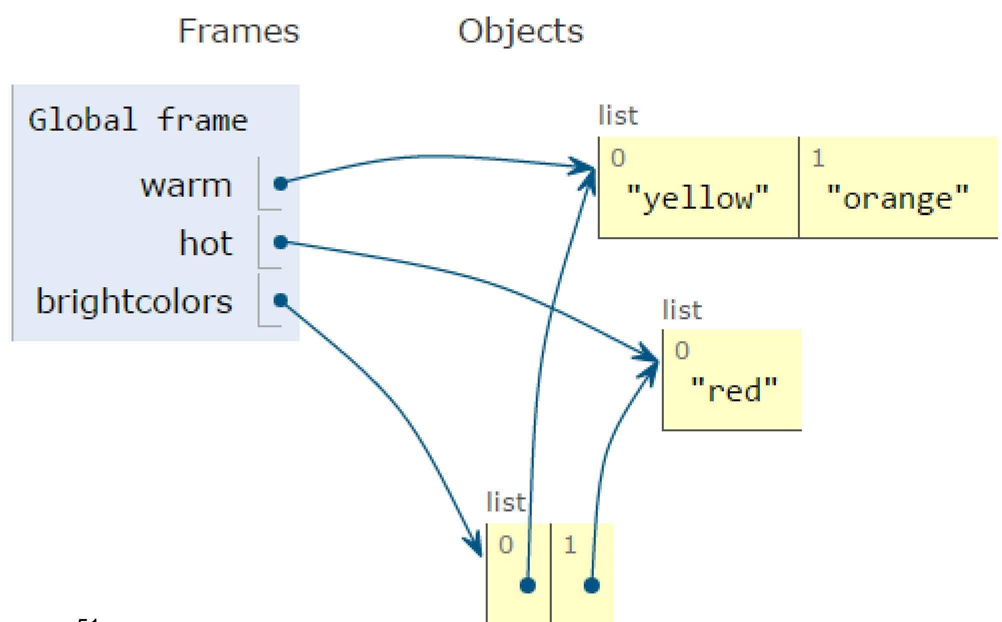


LISTS OF LISTS OF LISTS OF….

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

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

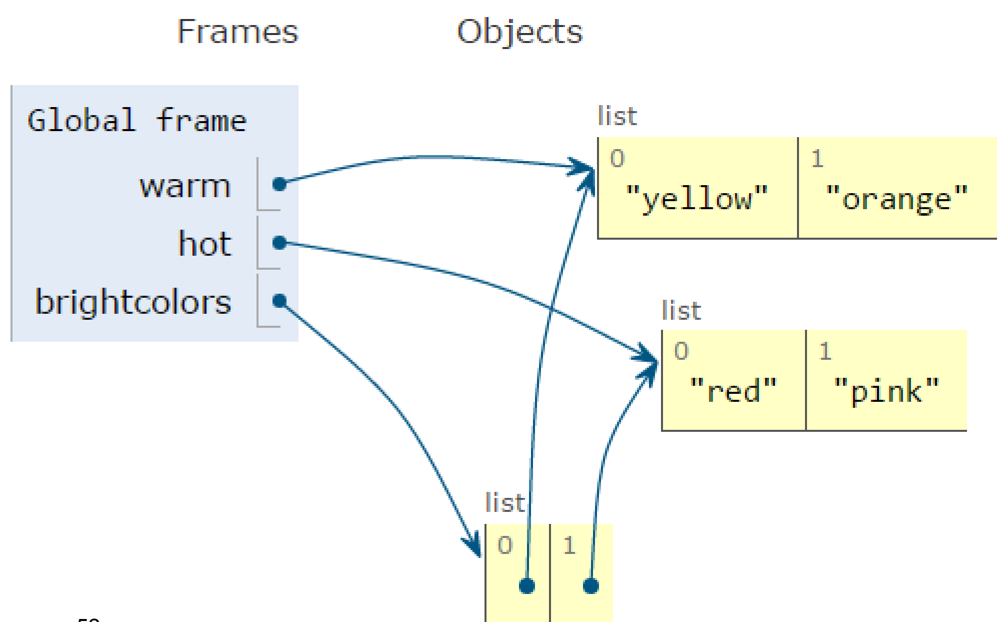
```python
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
```



LISTS OF LISTS OF LISTS OF….

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

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

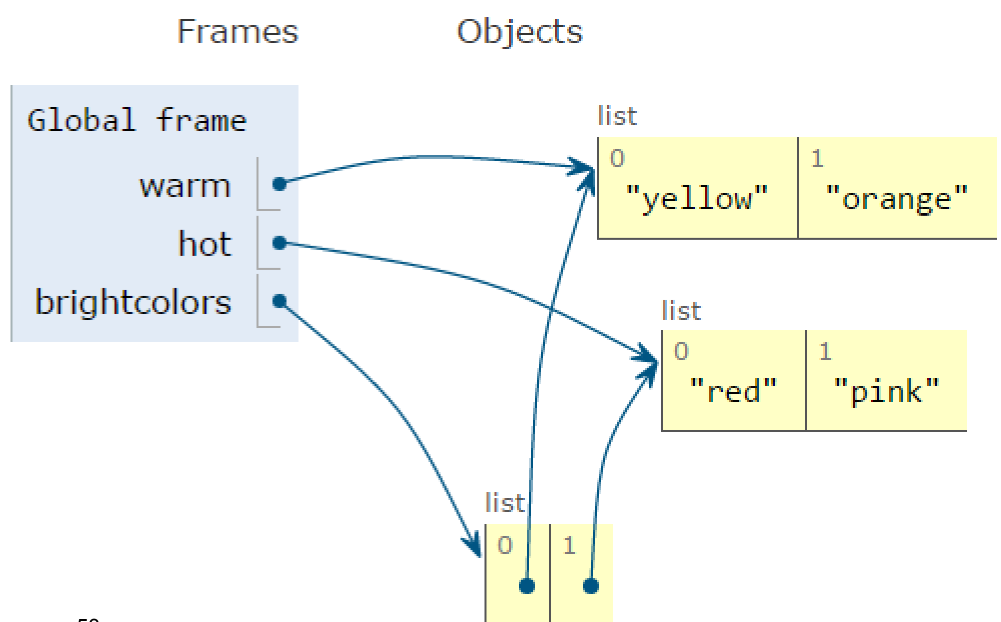



```python
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
hot.append('pink')
print(hot)
print(brightcolors)
```

LISTS OF LISTS OF LISTS OF….

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

```
[['yellow', 'orange'], ['red']]
['red', 'pink']
[['yellow', 'orange'], ['red', 'pink']]
```




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