CS1210 Lecture 13 Sep. 22, 2021

- Quiz 1 scores will take a few more days - sorry for the slow feedback
- HW3 due Friday
- DS4 is available, due 8pm tonight

Last time

- For-while conversion
- More on lists: examples, and mutability and aliasing
- + vs append
- introduction to DS4

Today

- Related to DS4: largest anagram set
- Finish lists
  - More on aliasing
  - is operator and object identify (vs ==)
  - Lists as arguments to functions
- review of HW3, Q1 hints

Discussion section 4 example

What if we wanted to find the largest set of anagrams?

- Simple direct approach:

```
biggestAnagramList = []
for word in wordList:
    anagramList = findAnagramsOf(word, wordList)
    if len(anagramList) > len(biggestAnagramList):
        biggestAnagramList = anagramList
```

  Works okay for a couple thousand words (word5.txt) but too slow for large word sets like wordsMany.txt

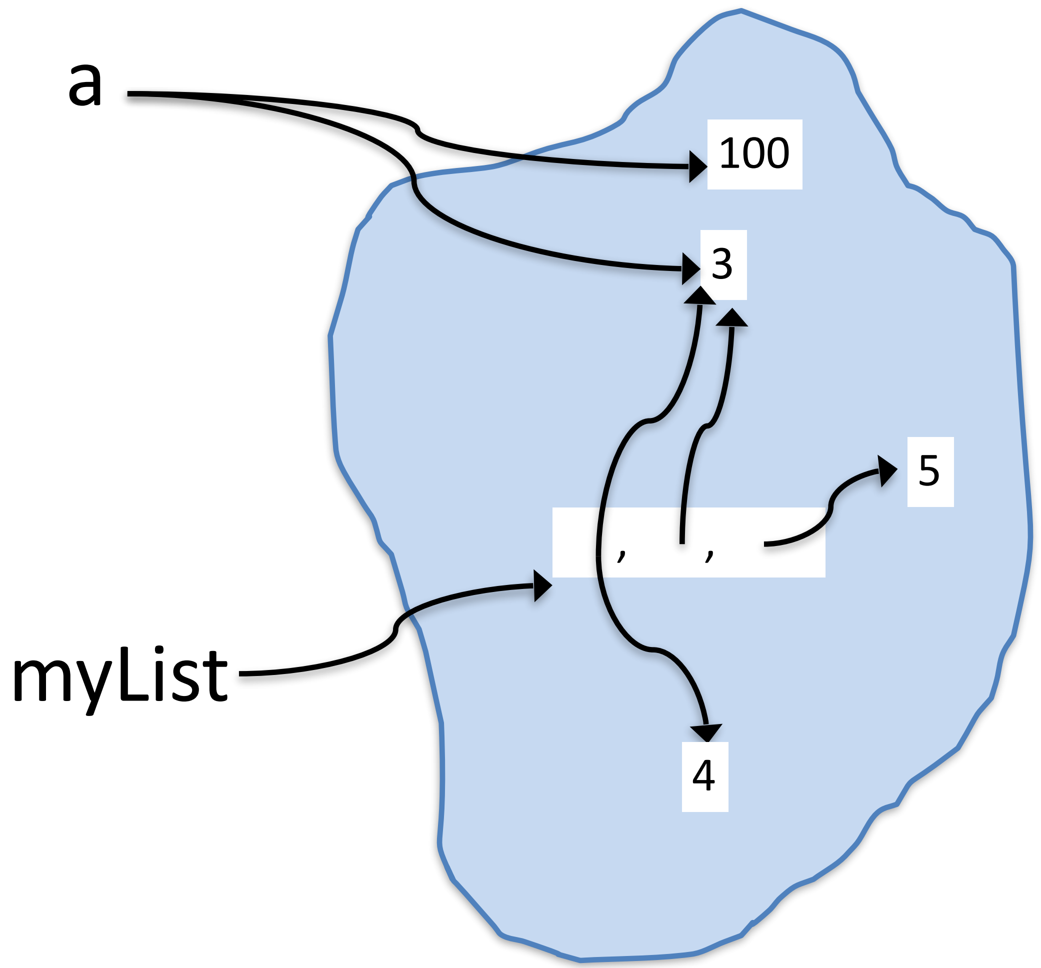
- Much faster approach: we'll look at this in detail in a couple weeks when we discuss algorithm analysis but, for now, the idea:

1. associate a "key" with each word, the sorted version of that word. E.g. ["art", "art"] … ["least", "aelst"] … ["rat", "art"] … ["stale", "aelst"] … ["tar", "art"]
2. Sort this list of pairs by those "keys". Now all anagrams are neighbors in this sorted list and the largest set can be found via one simple scan through it. […. ["least", "aelst"], ["stale", "aelst"], ..., ["art", "art"], ["rat", "art"], ["tar", "art"] ...]

biggestAnagramSet.py

List mutability

```
>>> a = 3
>>> myList = [a, a, 5]
>>> myList[0] = 4
>>> a = 100

>>>myList
???
```

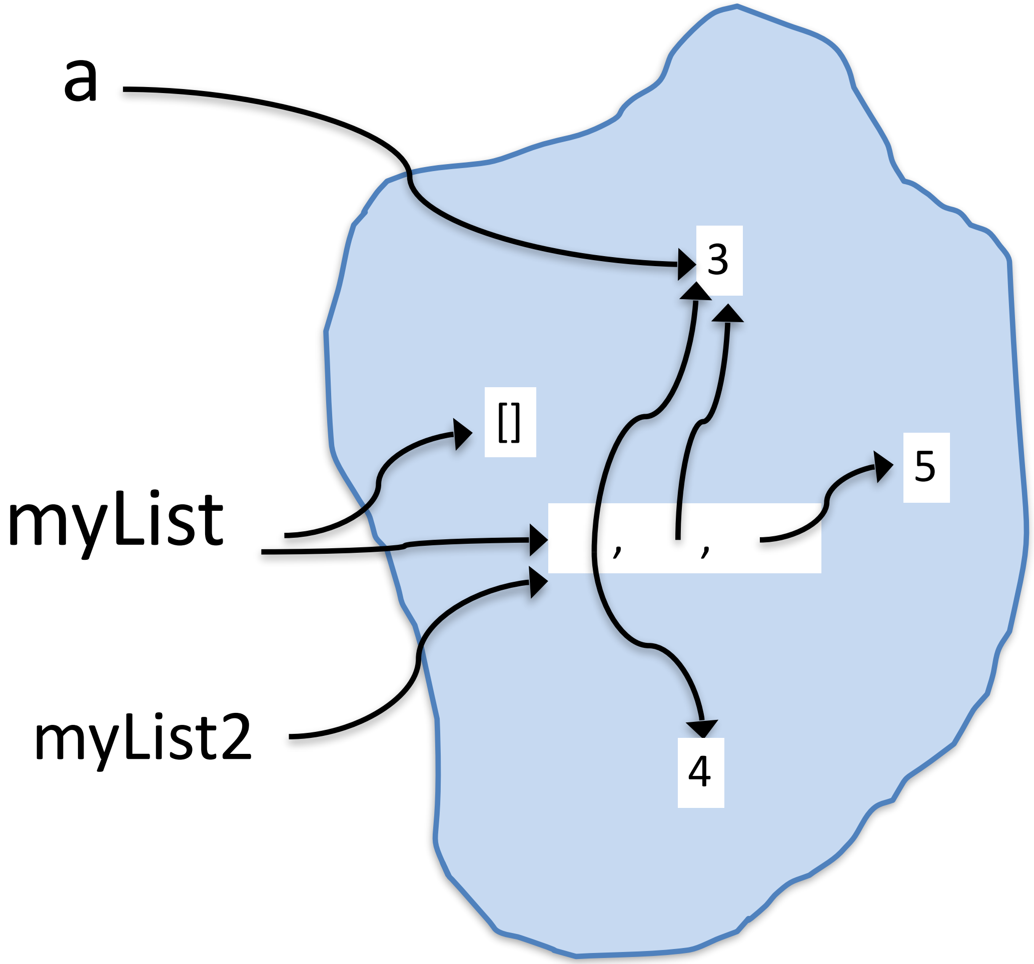


myList[0] = 4 *does not affect a's value!*

a = 100 does not affect list*!*

What happens here? Can you draw the updates?

```
>>> a = 3
>>> myList = [a, a, 5]
>>> myList2 = myList
>>> myList[0] = 4
>>>myList
???
[4, 3, 5]
>>>myList2
???
[4, 3, 5]
>>> myList = []
>>> myList
[]
>>> myList2
???
[4, 3, 5]
```



myList[0] = 4

- does not affect a's value!

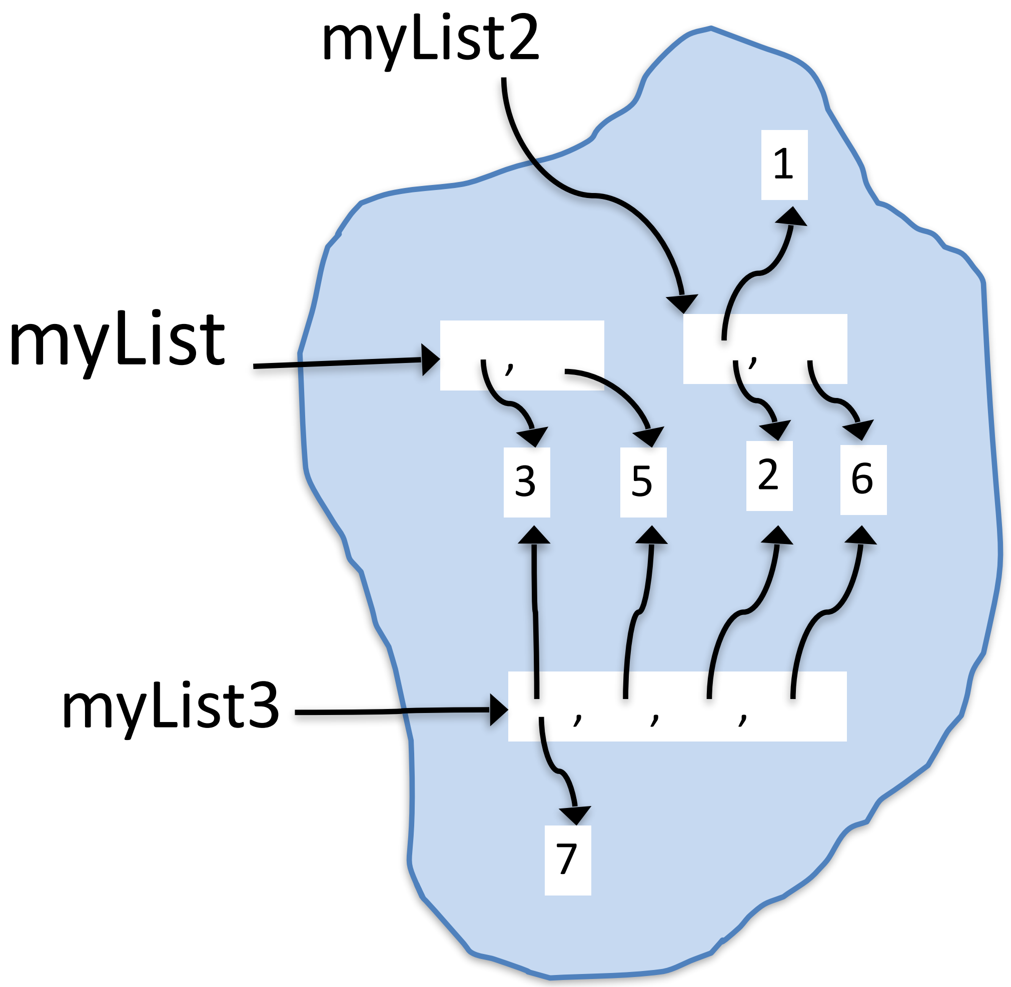
- does *affect myList2's value*

VERY IMPORTANT! CAN BE CONFUSING!

This is called **aliasing** – two or more variables referring to same mutable object

list +

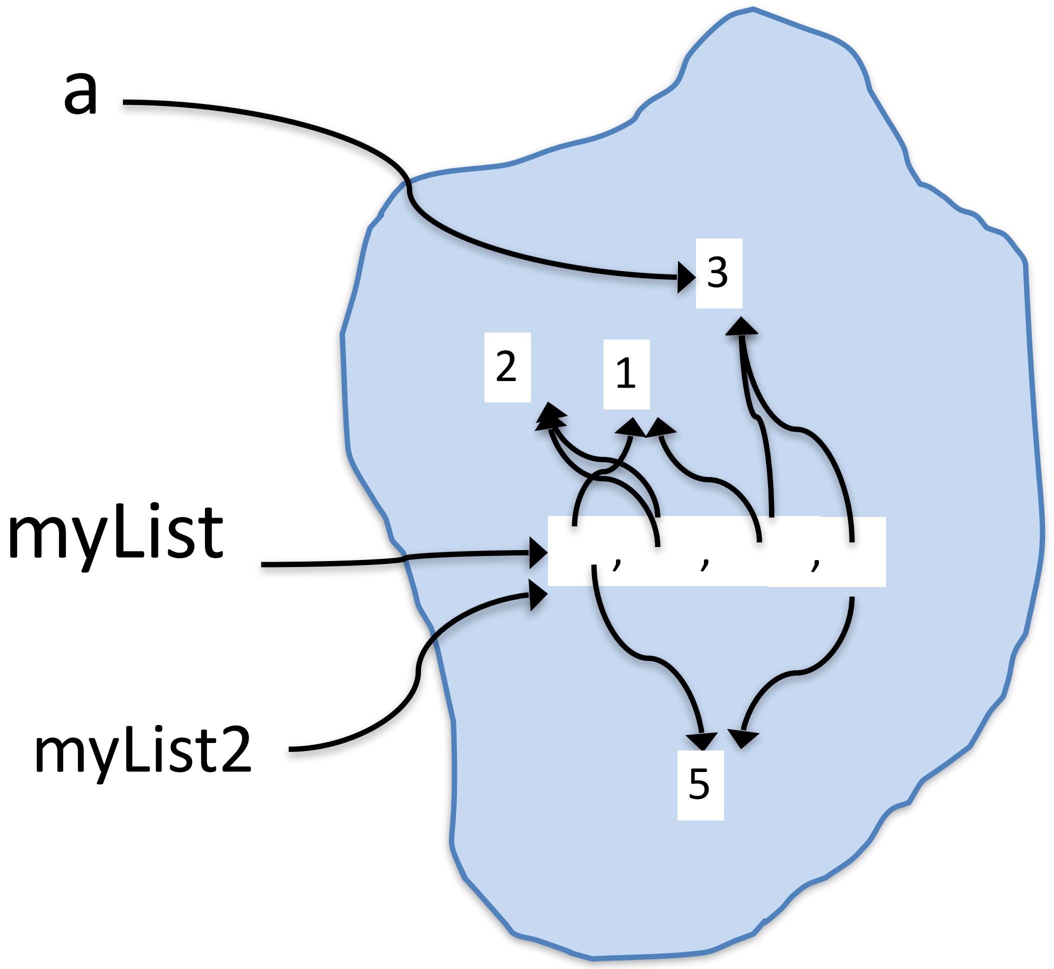
```
>>> myList = [3, 5]
>>> myList2 = [2, 6]
>>> myList3 = myList + myList2
>>> myList3
[3, 5, 2, 6]
>>> myList2[0] = 1
>>> myList3[0] = 7
>>> myList
?
>>> myList2
?
>>> myList3
?
```



IMPORTANT: + on lists yields a NEW list

append and sort

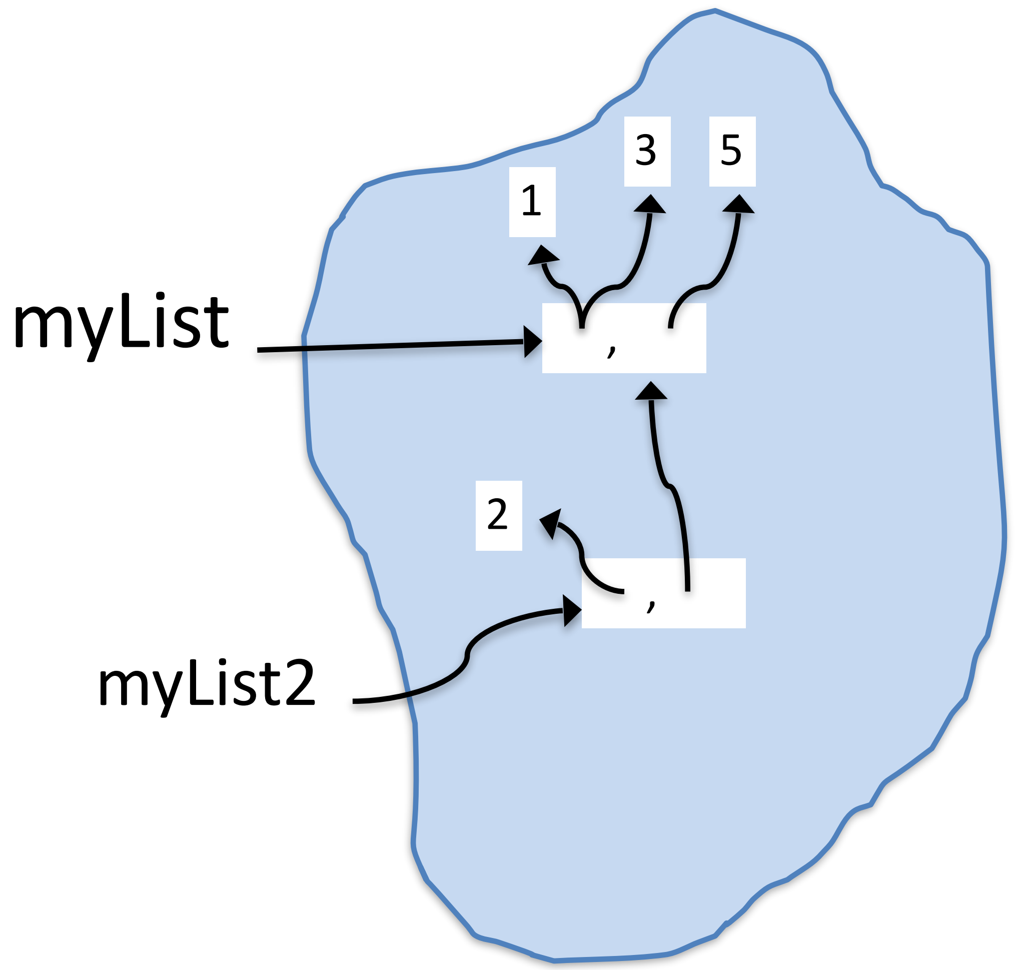
```
>>> a  = 3
>>> myList = [5, 2, 1]
>>> myList2 = myList
>>> myList.append(a)
>>> myList2.sort()
>>>myList
?
>>>myList2
???
```



SUPER IMPORTANT: unlike +, which does NOT modify the lists involved, append and sort MODIFY the list.

Consequences of list mutability

```
>>> myList = [3, 5]
>>> myList2 = [2, myList]
>>> myList[0] = 1
>>> myList2
?
```



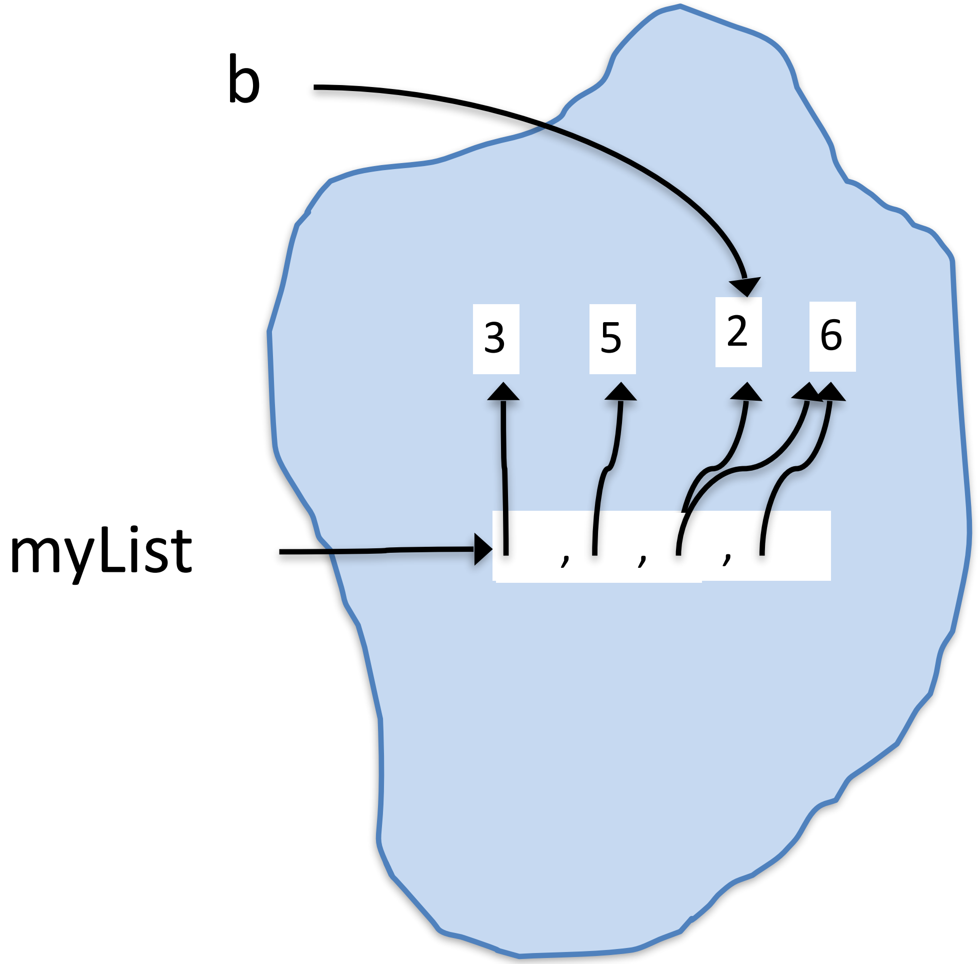
Important when we pass lists as arguments to functions!

del

del can be used to remove item or items from a list

```
>>> b = 2
>>> myList = [3, 5, b, 6]
>>> del myList[2]
>>> myList
[3, 5, 6]
```

- Can also **del** whole slices
- *I rarely need or use* del



Objects, equality, and identity

There is an operator in Python called **is**

```
>>> x is y
```

True if x and y refer to same object (in computer memory), False otherwise.

You don't often need to use **is** but you should be aware of when two variables refers to the same *mutable object.* This is called **aliasing.**

As we've seen:

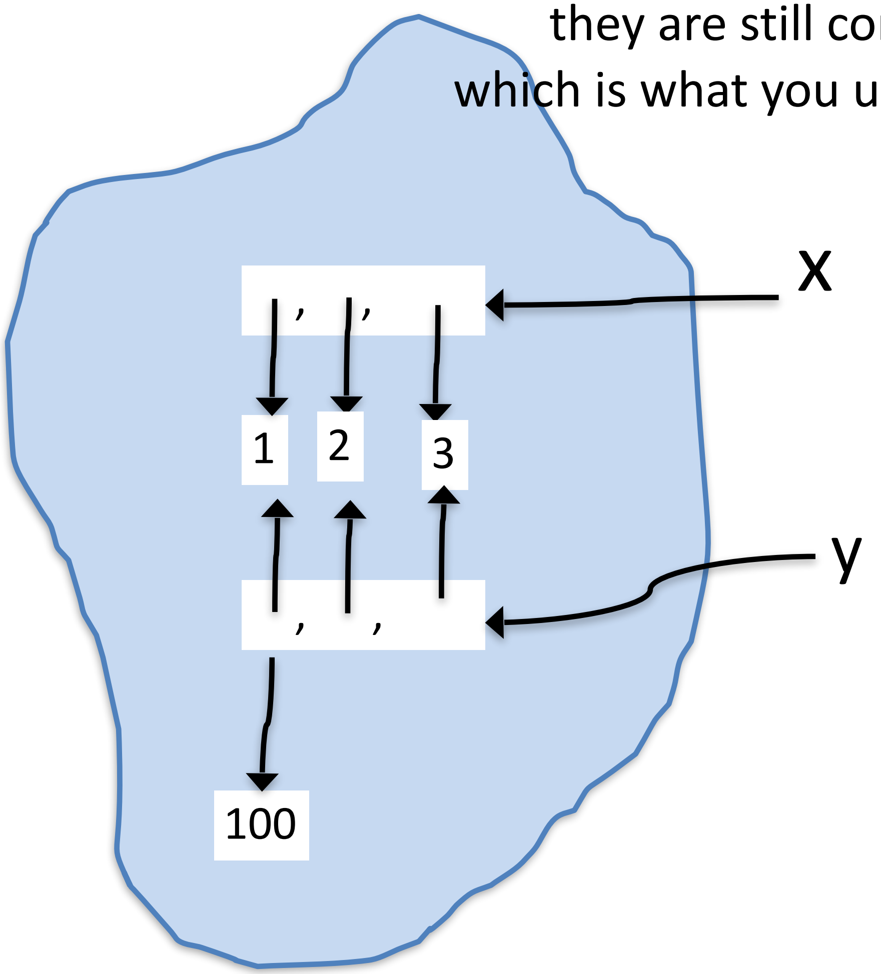
```
>>> x = [1,2,3]
>>> y = x
>>> x is y
True
>>> x[1] = 100
>>> y[1]
?
```

y and x are aliases for the same list object

Objects, equality, and identity

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> x is y
False
>>> x == y
True
>>> y[0] = 100
>>> x
???
```

constructs a list containing 1, 2, 3
constructs a (new/different) list
x, y are not aliases
they are bound to different objects
they are still considered equal, though, which is what you usually care about



Avoiding aliasing?

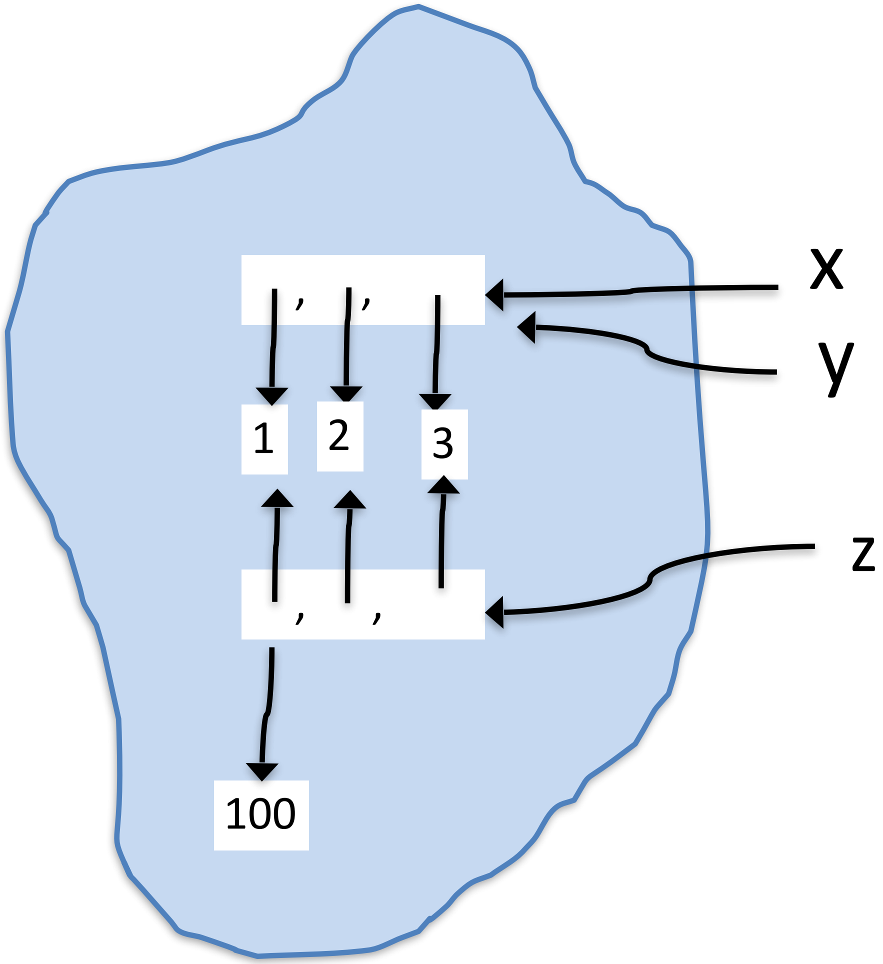
Often, we want to avoid aliasing. So, given a list, can we easily make a copy? YES!

```
>>> x = [1, 2, 3]
>>> y = x
>>> z = x[:]

>>> x is y
True
>>> x is z
False
>>> x == y
True
>>> x == z
True
>>> z[0] = 100
>>> y[0] = 50
>>> x
?
>>> y
?
```

range[:] is "full range" so a new list with all the elements of the original

Objects, equality, and identity

```
>>> x = [1, 2, 3]
>>> y = x
>>> z = x[:]

>>> z[0] = 100
>>> x
???
```



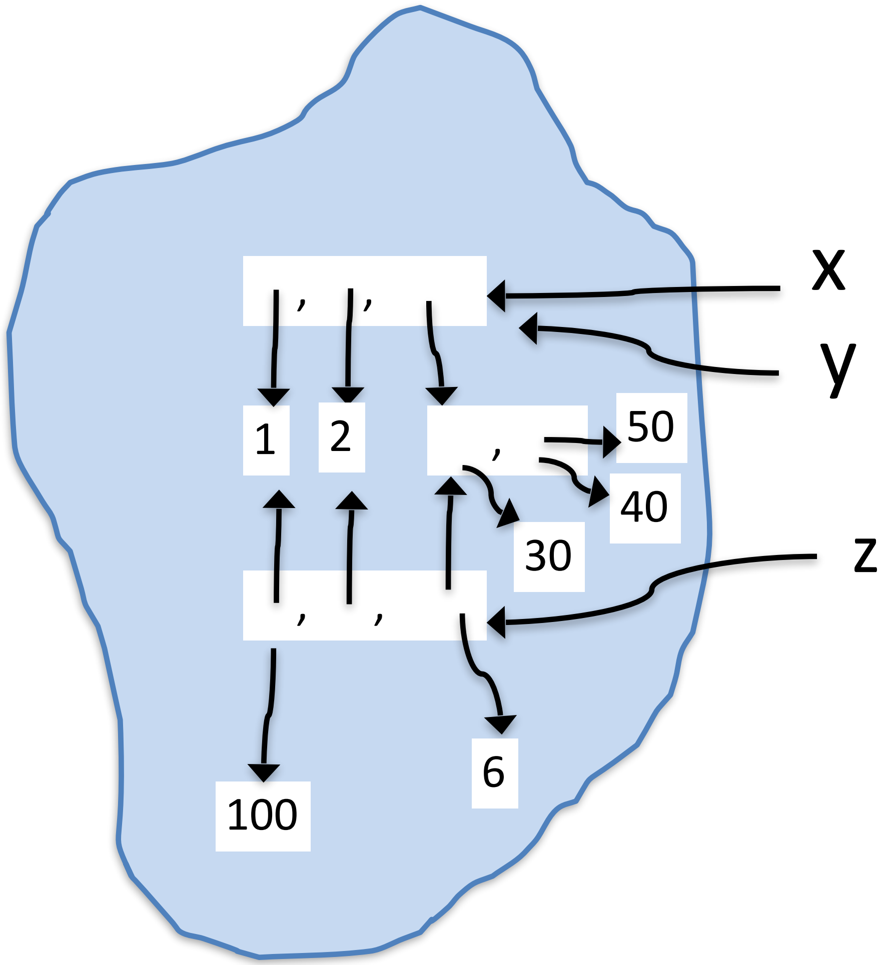
Objects, equality, and identity

But, be careful!

```
>>> x = [1, 2, [30, 40]]
>>> y = x
>>> z = x[:]
>>> x is y
True
>>> x is z
False

>>> z[0] = 100
>>> x
?
>>> z[2][1] = 50
>>> x
?
```

Objects, equality, and identity

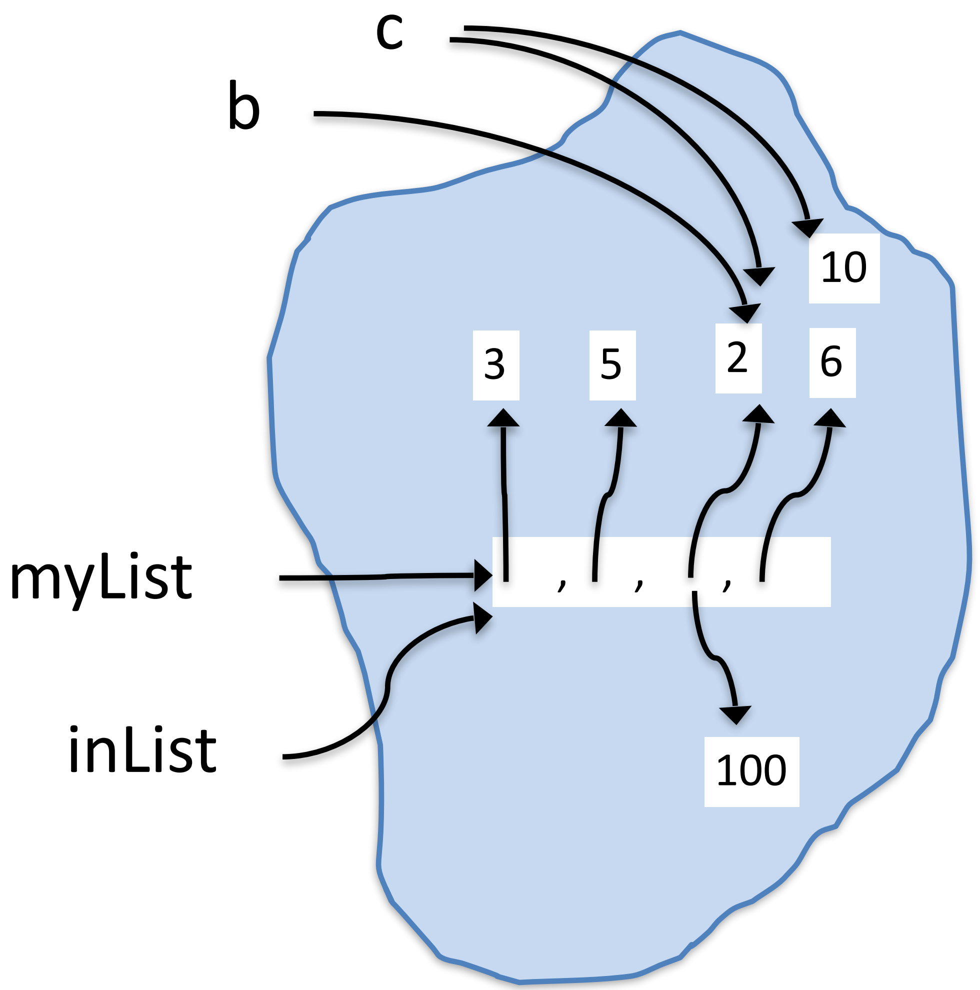
```
>>> x = [1, 2, [30, 40]]
>>> y = x
>>> z = x[:]

>>> z[0] = 100
>>> z[2][1] = 50
>>> x
???
>>> x
???
>>> Z[2] = 6
>>> z
???
>>>x
???
```



[:] is a *shallow* copy. There are ways to do *deep* copy (maybe we will discuss later in the semester)

Mutability and arguments to functions

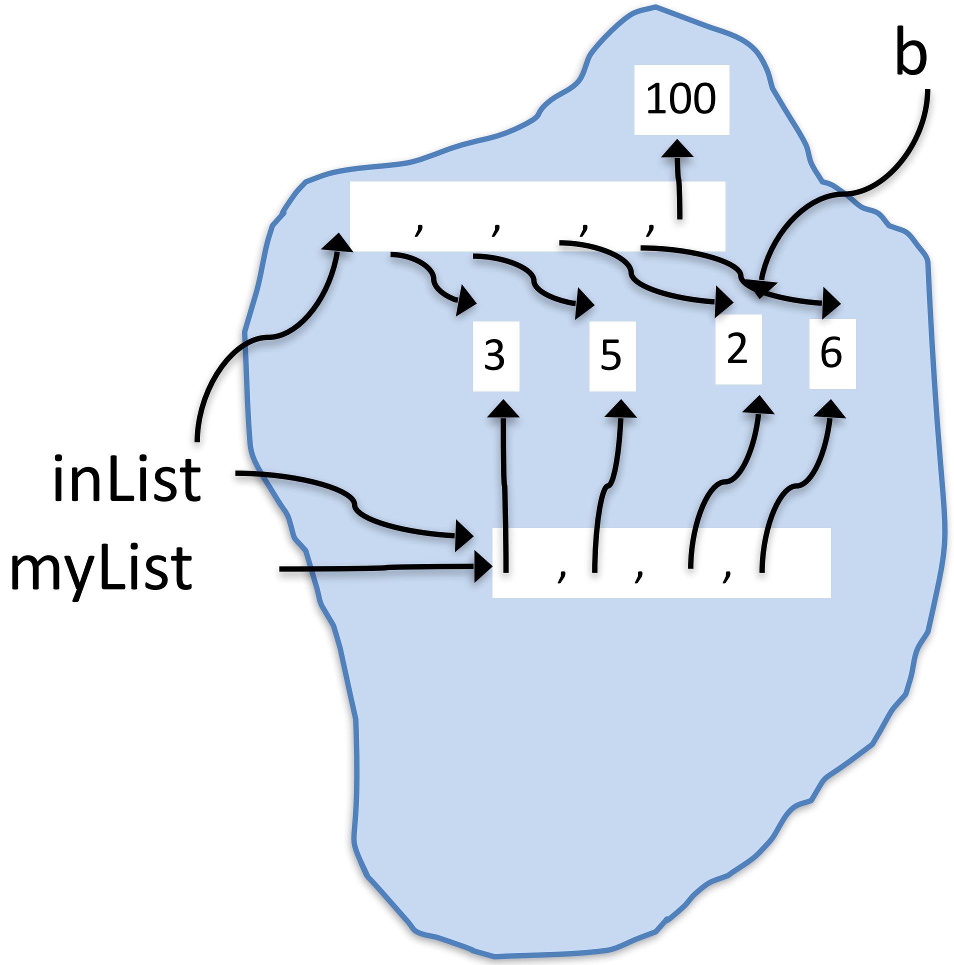
```
>>> def foo(inList, c)
...         inList[2] = 100
...          c = 10
>>>
>>> b = 2
>>> myList = [3, 5, b, 6]
>>> foo(myList, b)
>>> myList
[3, 5, 100, 6]
```



But what if body of foo is instead: inList = inList + [10]?

Mutability and arguments to functions

```
>>> def foo2(inList)
...         inList = inList + [100]
>>>
>>> b = 2
>>> myList = [3, 5, b, 6]
>>> foo2(myList)
>>> myList
[3, 5, 2, 6]
>>> b
```



Advice/comments on functions

- Some functions compute something and return a value without *side effects*. That is, they do any output and don’t change the values of any objects that exist outside of the execution of that function.
- Other functions do have side effects. They either print something (or affect GUI elements) or change values of objects that exist outside the function execution. Such functions often don’t return anything. And such functions can maybe helpfully be thought of as commands.

```
# return new list that is like inList
# but without 1st and last elements
def middle(inList):
    return inList[1:len(inList)-1]
```

```
# remove the first and last
# elements from inList
def chop(inList)
    del inList[0]
    del inList[len(inList)-1]
```

We use these differently.
Consider:

```
def bar(inList):
    …
    middle(inList)
    …
    …
```

What can you say about this?

And

```
def baz(inList):
    …
    chop(inList)
    …
    ..
```

And how about this?

Look at the code in lec13.py and make sure you understand the differences between bar, bar2, and baz

Problem like HW3 Q1

Suppose goal is to find second and third smallest letters, and most common letter

A two-part approach (you *can* do it "all at once" if you want but many people will find separating the two easier):

```
# find second and third smallest
        # go through string char by char updating values for
        # three simple variables:
        #   smallest, secondSmallest, and thirdSmallest

# find most common
        # presume you have a function howMany(c, s) that
        #       returns the number of times c occurs in s
        # Using a loop simply go through string char by char,
        #       calling howMany(char, s) for each char and comparing result with a   #
maxOccurrencesSoFar variable, updating when appropriate

# print results
```

howMany(c, s)
is easy to write!

Hint: consider using **None** for initializing variables

HW3 Q1

```
# find second and third smallest
    # go through string char by char updating values for
    # three simple variables:
    #    smallest, secondSmallest, and thirdSmallest
```

smallest:	~~?~~	~~e~~	~~c~~	~~b~~	a
secondSmallest:	~~?~~	~~e~~	~~d~~	~~c~~	b
thirdSmallest:	~~?~~	~~e~~	~~d~~	c	

e c d b f a

Next Time

- Start Ch 12 - dictionaries