CS2110 Lecture 12 Feb. 19, 2021

- HW3 due Monday
- DS4 assignment document will be made available tonight (but finish HW3 first! It's more important and harder.)

Last time

- Iteration over lists
- list mutability

Today

- For-while conversion
- More on list mutability and aliasing
- + vs append
- **is** operator and object identify (vs ==)
- Review of HW3, Q1 hints, and introduction to DS4

for -> while conversion

```
for var in sequence:
    ...
    ...
    ...
```

```
index = 0
while index < len(sequence):
    var = sequence[index]
    ...
    ...
    ...
    index = index + 1
```

Completely mechanical. No thought needed.
Body (the … lines) ***does not change***.

lec12forwhile.py

(last time) Ch 10: lists are mutable!

- Strings are immutable. You can't change them.

```
>>> myString = 'hello'
>>> myString[0] = 'j'
```

← Error

- But lists are mutable! You can update lists

```
>>> myList = [1, 2, 'hello', 9]

>>> myList[1] = 53
 >>> myList
[1, 53, 'hello', 9]
```

you can replace a item in a list with a new value

```
>>> myList.append('goodbye')
>>> myList
[1, 53, 'hello', 9, 'goodbye']
>>> myList = myList.append(3)
```

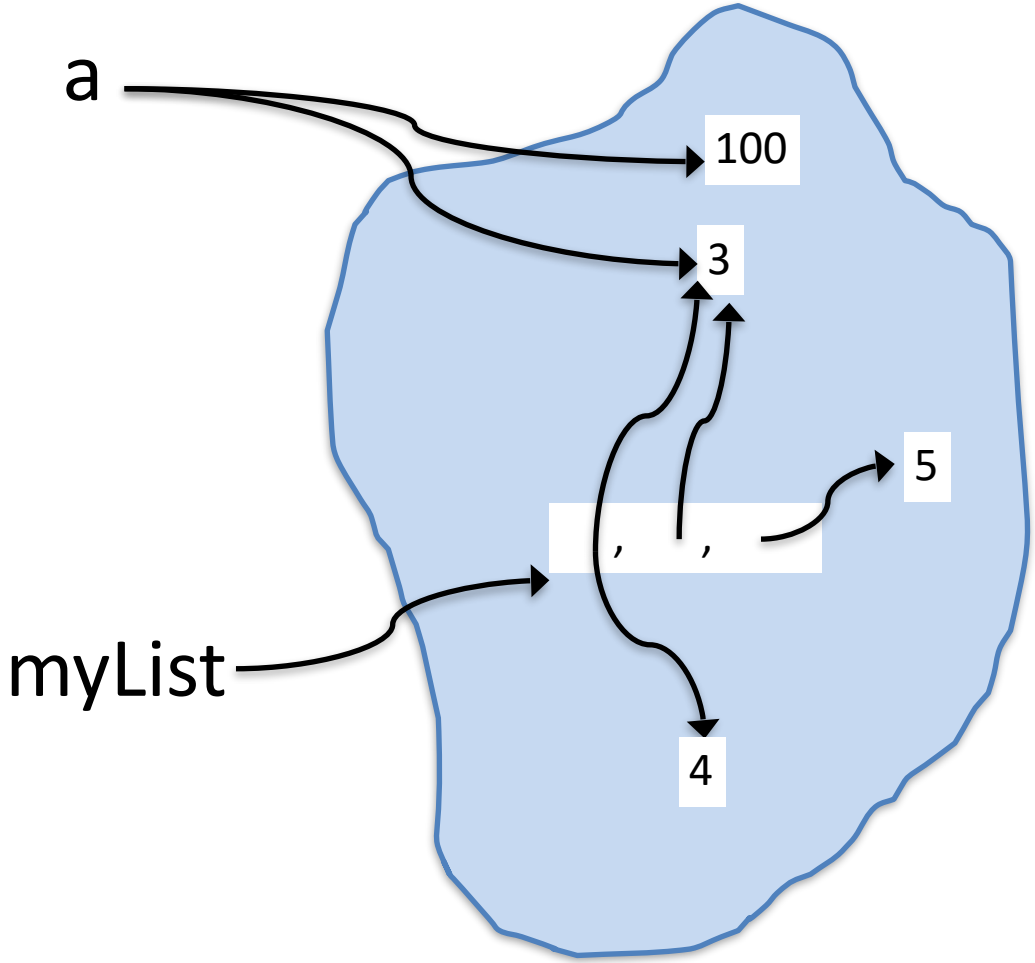
you can add new items to the end of a list

```
>>> myList2 = [3, 99, 1, 4]
>>> myList2.sort()
>>> myList2
[1, 3, 4, 99]
```

you can even sort! Note: Python's sort rearranges the items directly within the given list. It doesn't yield a new list with same items in sorted order (different function, sorted, yields new sorted list)

List mutability

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

>>>myList
???
```

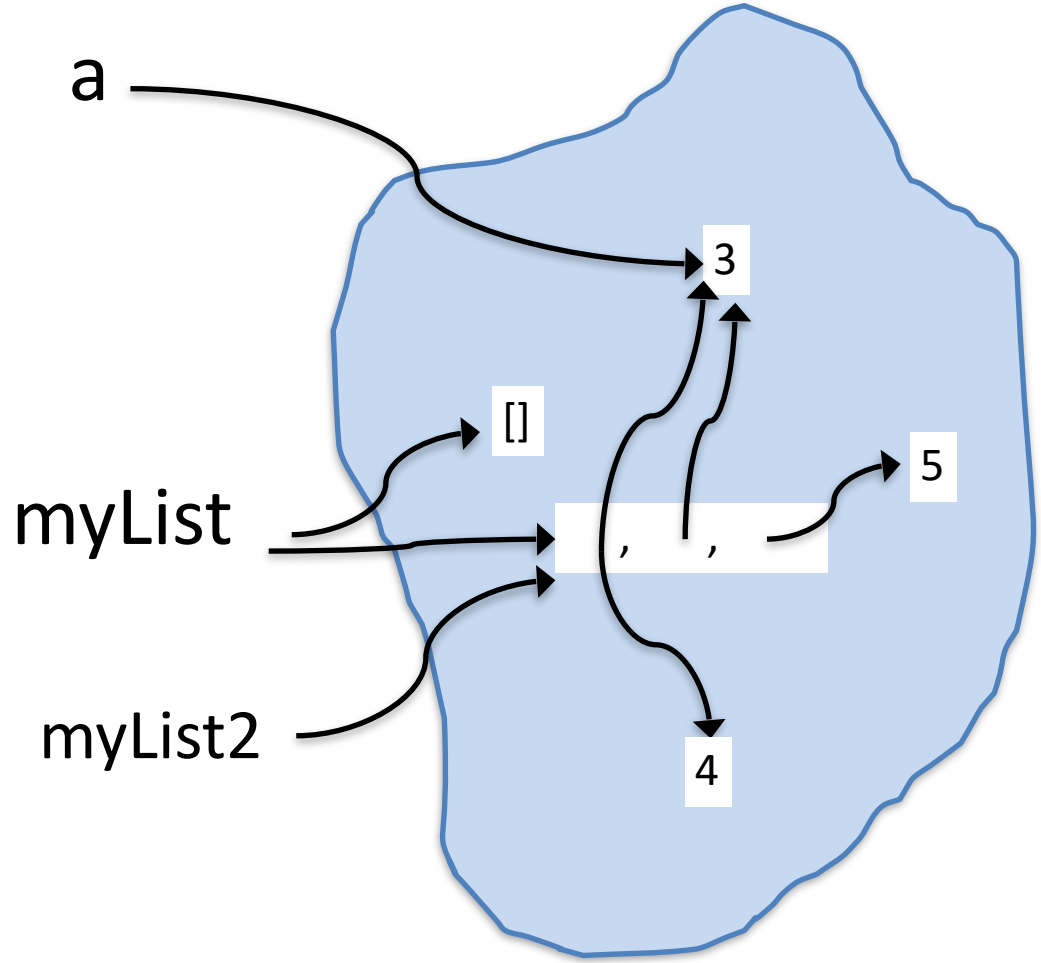


myList[0] = 4 *does not affect a's value!*

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

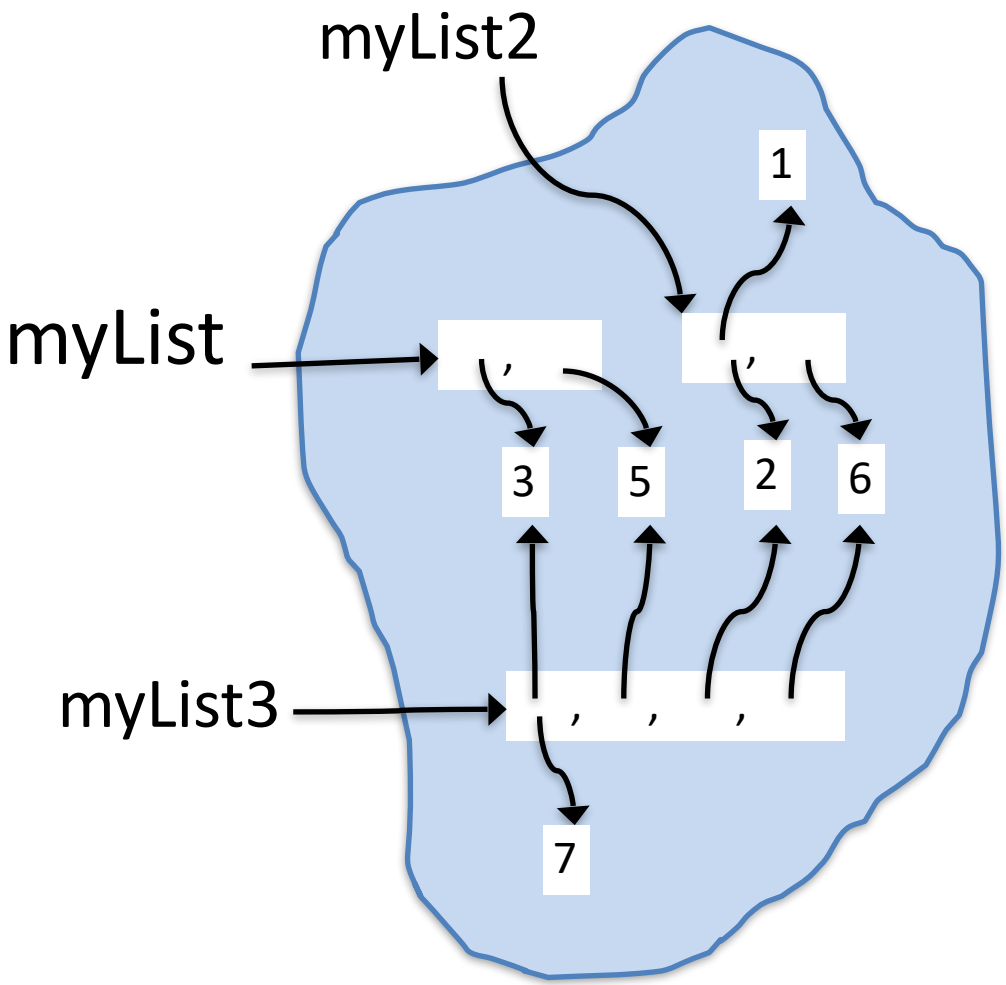
- *does not affect a's value!*
- does *affect myList2's value*

VERY IMPORTANT! CAN BE CONFUSING!

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

list +

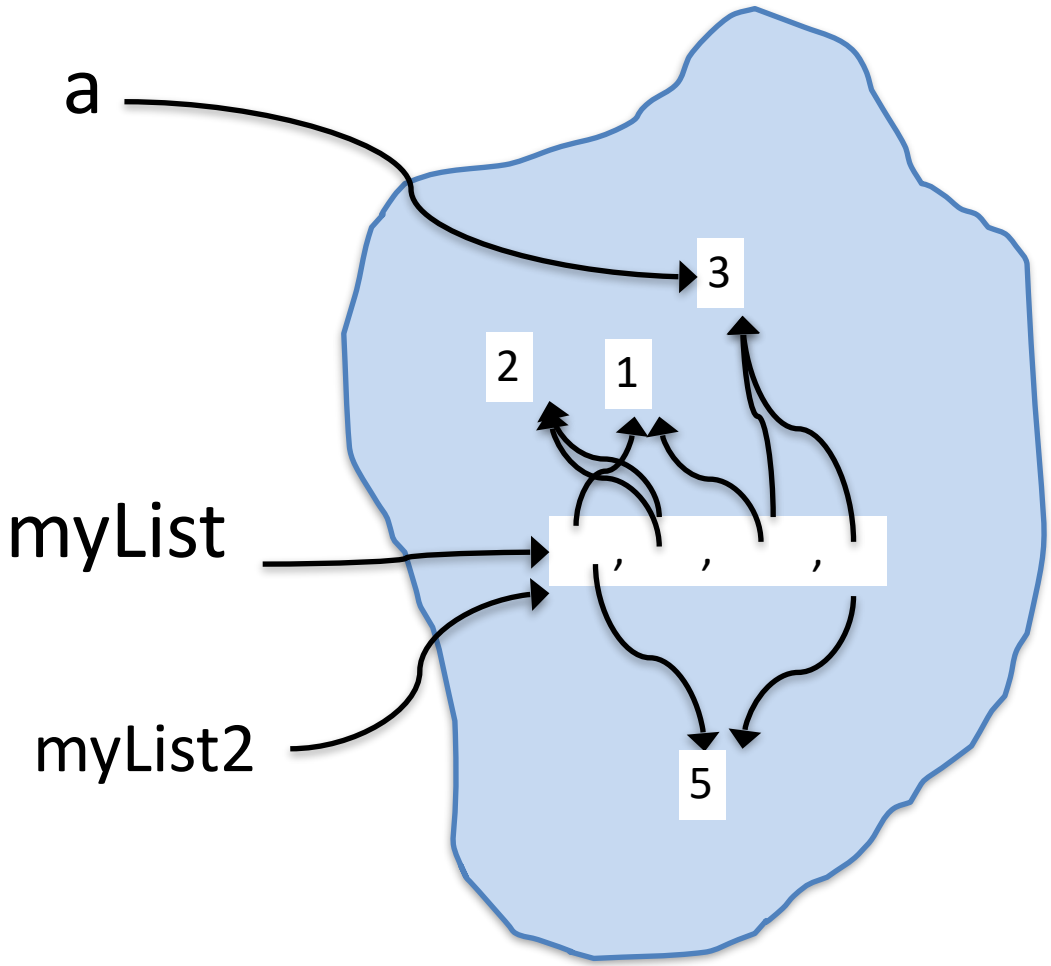
```
>>> myList = [3, 5]
>>> myList2 = [2, 6]
>>> myList3 = myList + myList2
>>> myList3
[3, 5, 2, 6]
>>> myList2[0] = 1
>>> myList3[0] = 7
>>> myList
?
>>> myList2
?
>>> myList3
?
```



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.

list + *vs.* append

```
result = []
for num in range(100000):
    result = result + [num*num]
```

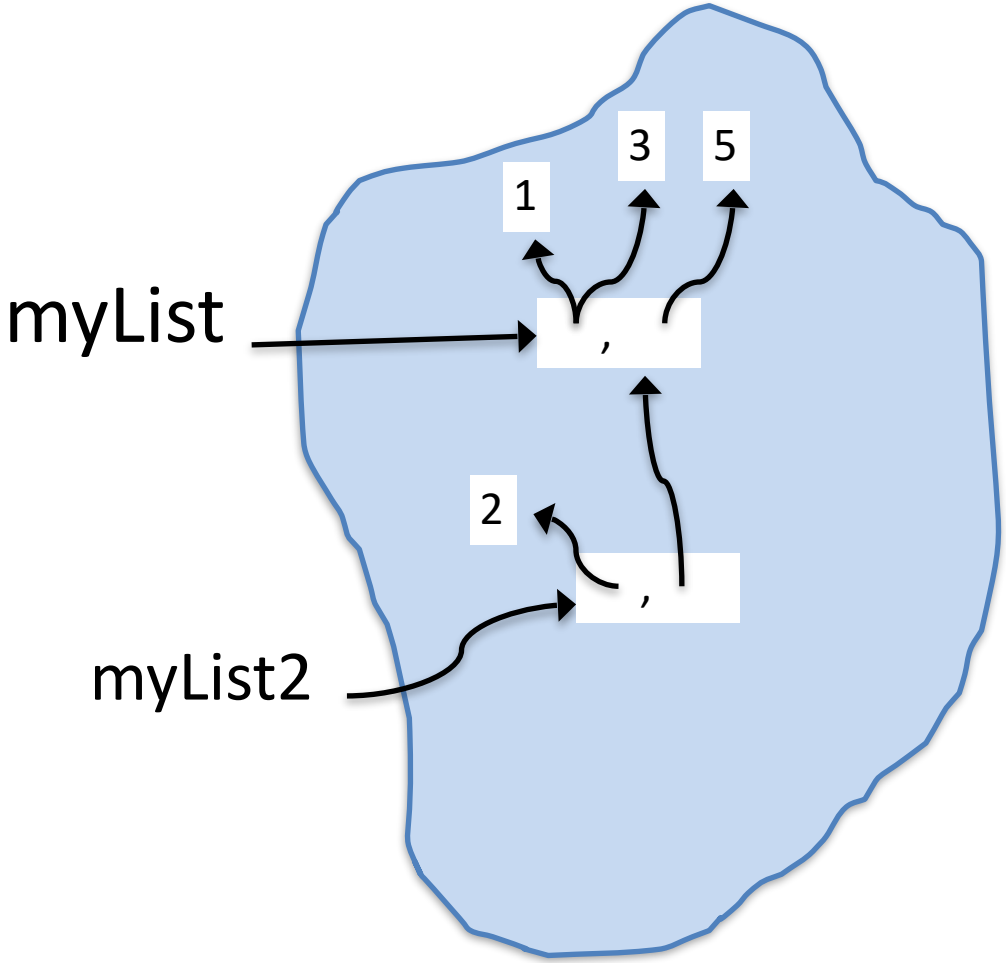
```
result = []
for num in range(100000):
    result.append(num*num)
```

Is either one better?

lec12append.py

Consequences of list mutability

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



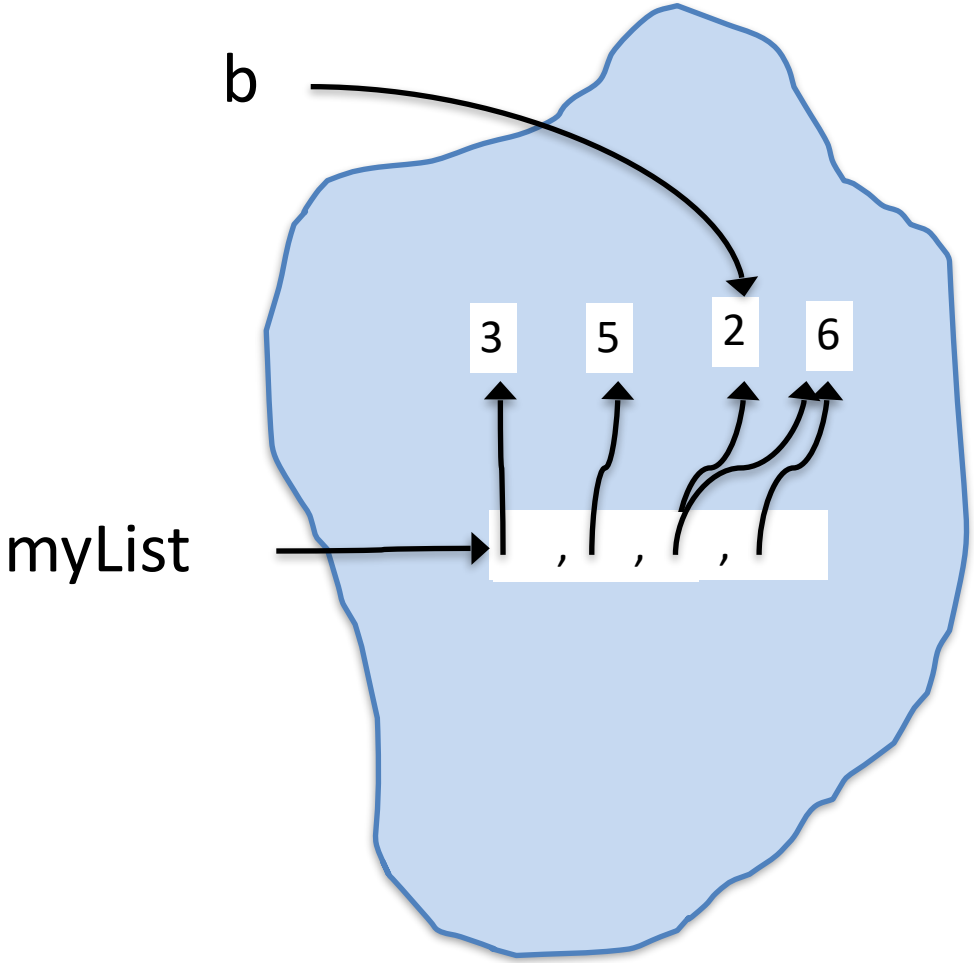
Important when we pass lists as arguments to functions! (next lecture)

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:

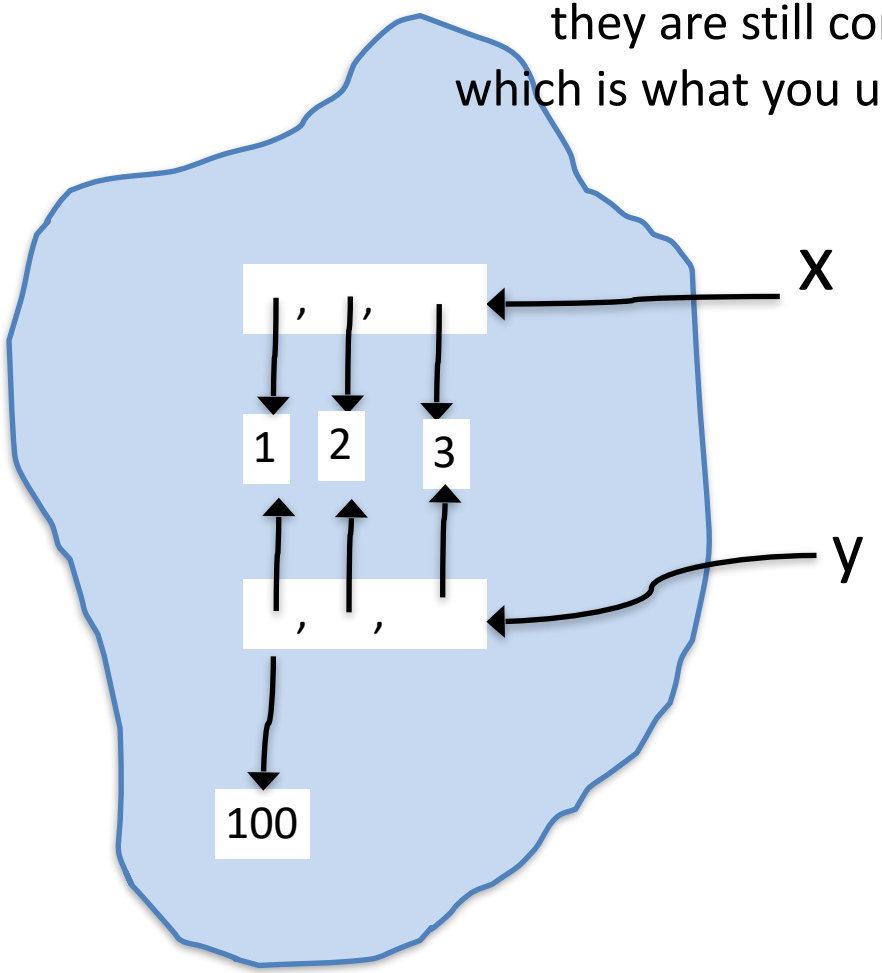
```
>>> x = [1,2,3]
>>> y = x
>>> x is y
True
>>> x[1] = 100
>>> y[1]
?
```

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?

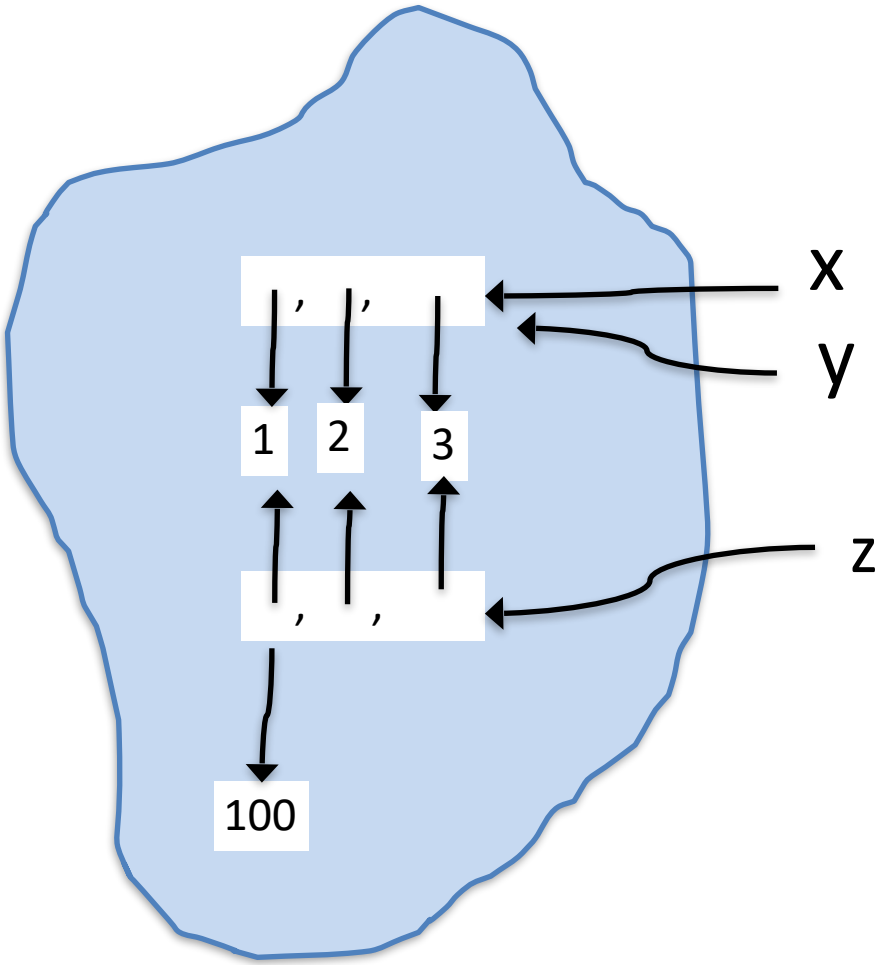
Often, we want to avoid aliasing. So, given a list, can we easily make a copy? YES!

```
>>> 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
???
```



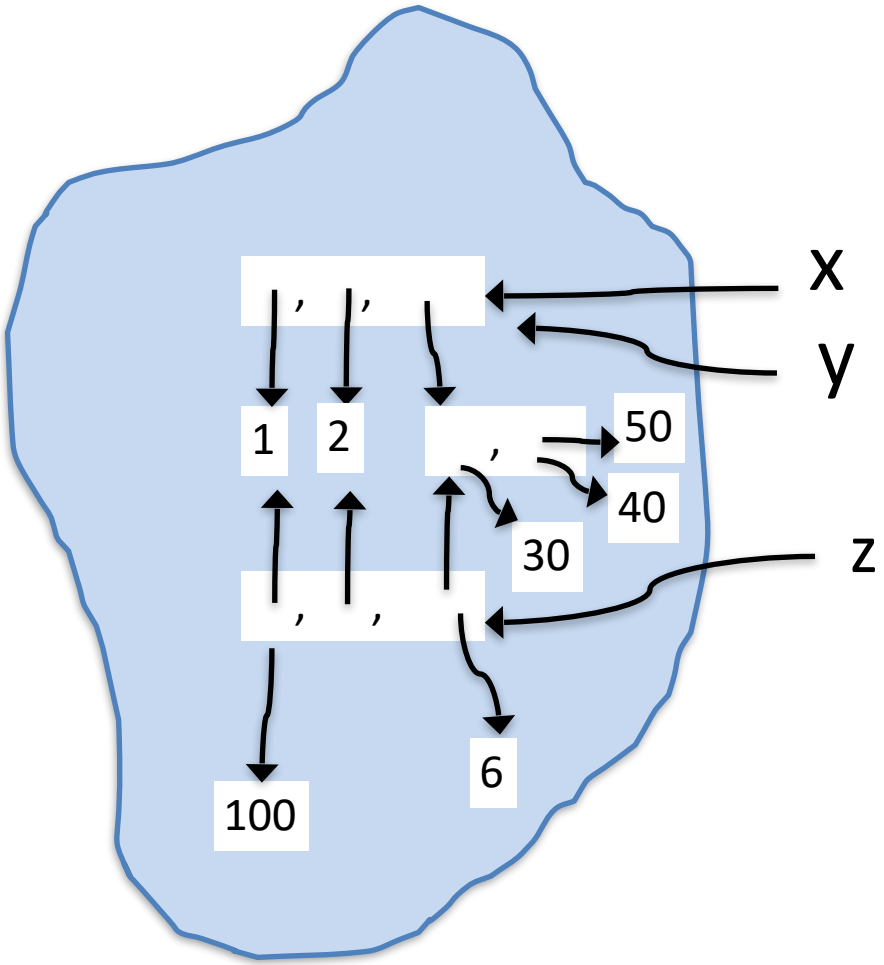
Objects, equality, and identity

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)

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

Discussion section 4

- Will work with files of many words and write code to find sets of anagrams (words with same letters but different order). E.g. art, rat, tar
- What if we wanted to find the largest set of anagrams?
  - simple direct approach

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

  - Works okay for a couple thousand words (words5.txt) but far too slow for 100+K list like wordsMany.txt
  - Problem to think about: can you efficiently find largest anagram set?

biggestAnagramSet.py

Next Time

- One more part of Ch 10 – 10.22, list comprehensions
- Start Ch 12 - dictionaries