CS1 Lecture 12 Feb. 13, 2017

- HW4 available tomorrow.
- Discussion sections this week
  - will be closely tied to one of the homework problems.
  - for paired sections, new partners
- Exam 1, Thursday evening, 2/23, 6:30-8:00pm
- HW2 scores posted today
  - BUT a number of people will receive email from me (and changed HW scores) regarding academic dishonesty.
  - Check comments even if you got 10/10. Sometimes we make suggestions even for correct answers

Last time

Continued Ch 10

- **for** loops and lists
- lists are *mutable*
- two examples

Today

More Ch 10

- **for** loops on lists and **range**s
  - and for -> while conversion
- more on list mutability
  - + vs append
- more examples

(last time) Ch 10: traversing lists

Like they are for strings, **for** loops are again concise and useful for iteratively accessing each item of a list

```
for element in [‘a’, 2, ‘word’, [‘1,2’, 3]]:
        if type(element) == list:
            print(‘list of length:‘, len(element))
        else:
            print(element)
```

yields:

a

2

word

list of length: 2

Ch 10: **range**

The “Traversing a List” section is almost the only place where our textbook describe Python’s very commonly used **range** type. The **range** type is another sequence type, like **list** and **string**.

range(9) is a sequence of the integers 0, 1, …, 8

range(2,6) is sequence 2, 3, 4, 5

range(2,13,3) is sequence 2, 5, 8, 11

Since range is a sequence type, (most of) the standard sequence operations apply (not nicely specified any where in text – go to Python sequence docs on-line)

Ch 10: **range** – standard sequence ops

```
>>> 5 in range(9)
True
>>> 5 in range(2,10,2)
?
>>> len(range(2,10,2))
?
>>> myRange = range(2,20,2)
>>> myRange[3:6]
?
>>> range(5) + range(5)
?
```

Ch 10: **range** – Python 3 vs Python 2

In Python 2, range is just a function that produces a list:

```
>>> range(9)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

In Python3, range(9) is an object that represents the same sequence of numbers, but it *not* a list.

```
>>> range(9)
range(9)
```

Note: in Python 3, you can still use range to build an ordered list of numbers:

```
>>> list(range(9))
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Ch 10: **range** and **for**

So, why does the text introduce **range** in the "Traversing a List" section? Because the book was originally written for Python 2 when range produced a list, and because **for** and **range** are very commonly used together to iterate over a range of numbers. E.g.:

```
for num in range(1000):
    print(num*num)
```

Question: why might Python 3's range function might be better than using lists/Python 2's range for things like this?

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']
```

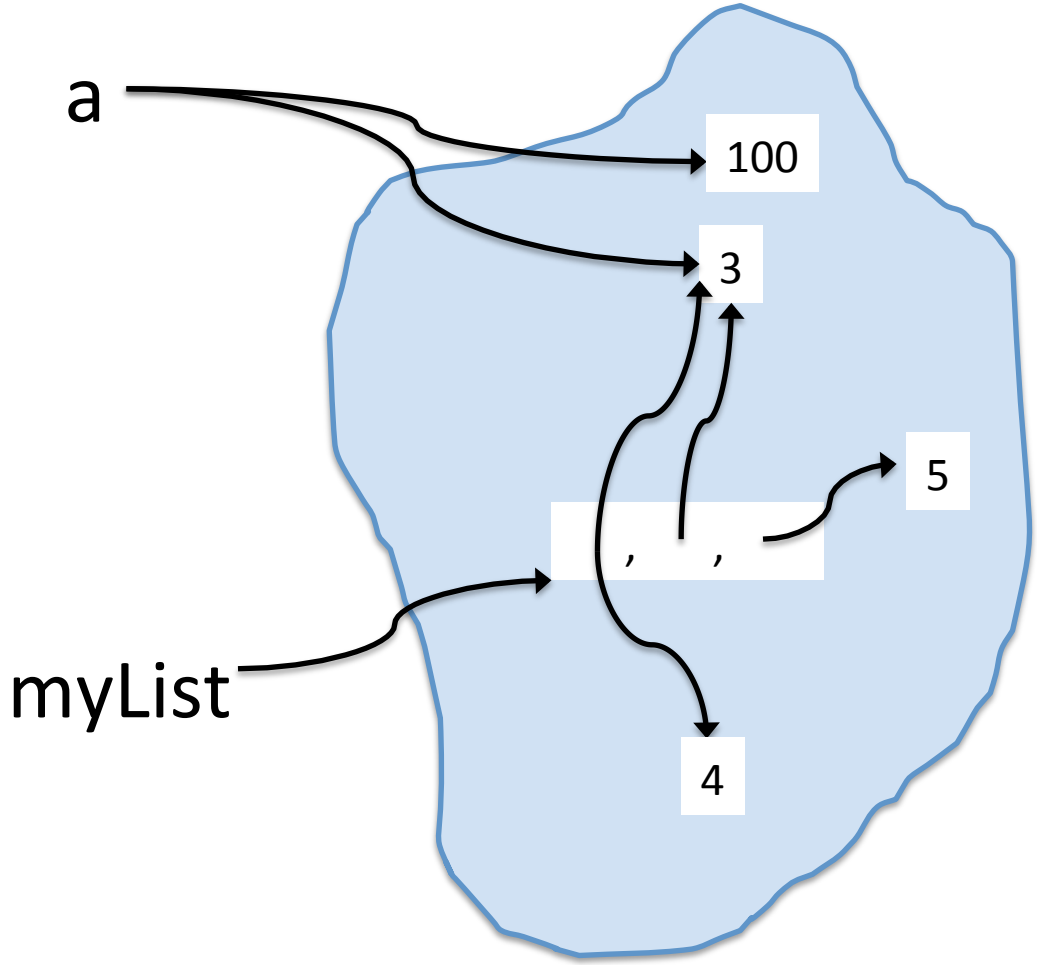
you can add new items to the end of a list

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

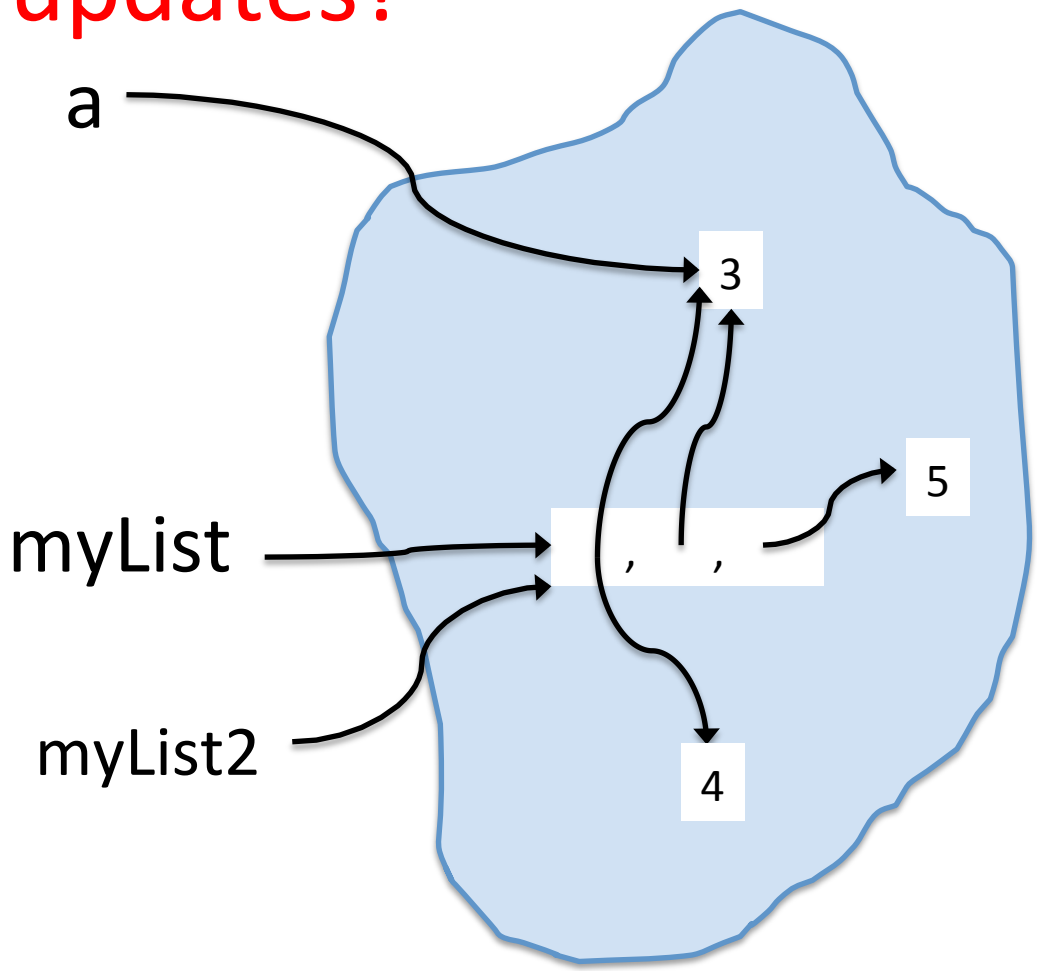


myList[0] = 4 *does not affect a's value!*

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
???
```



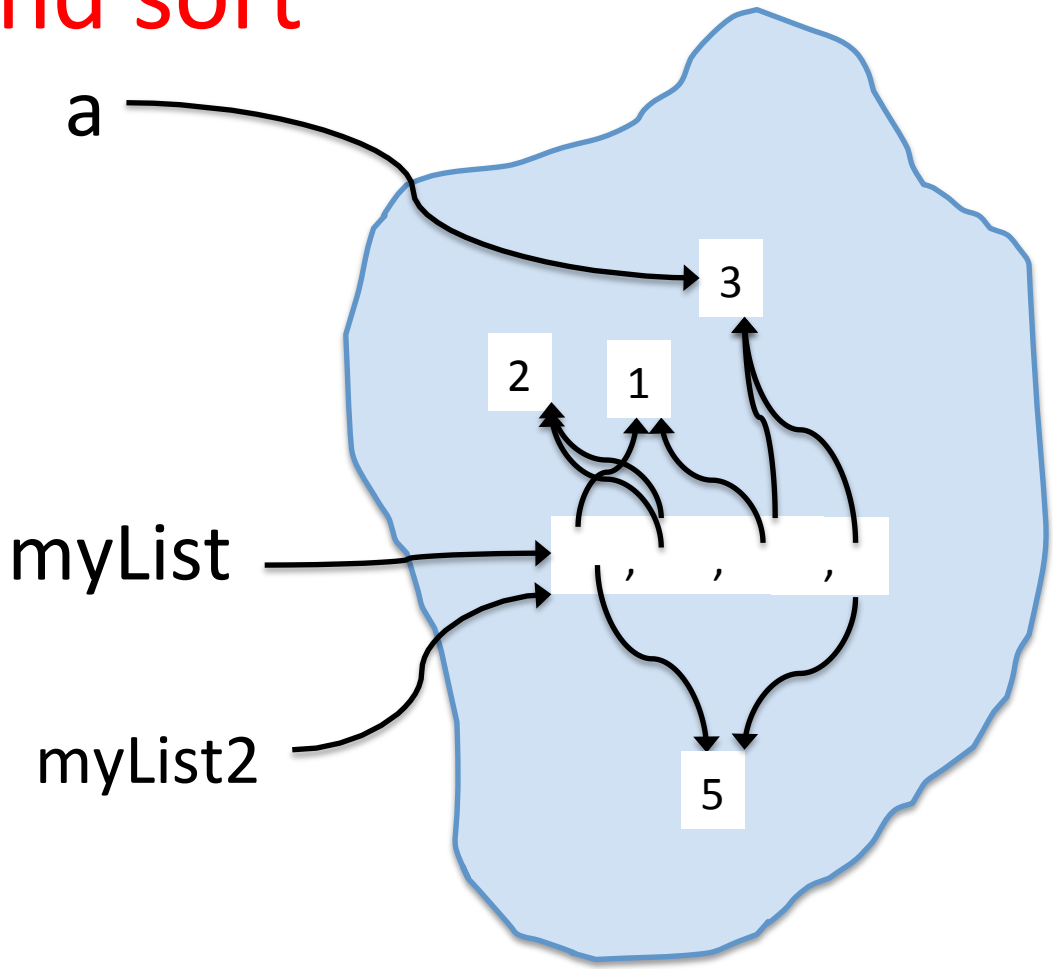
myList[0] = 4

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

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

Similarly with operations like append and sort

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



list + *vs.* append

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

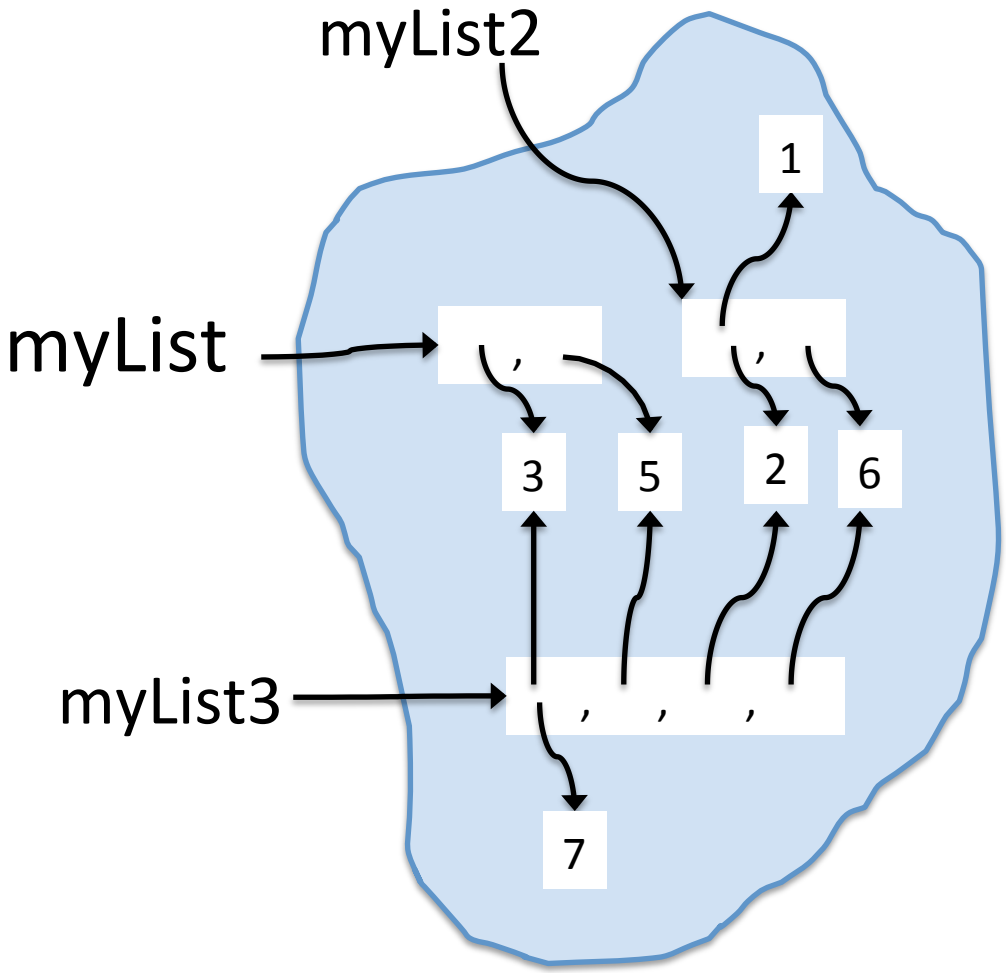
```
result = []
for num in range(100000):
    result.append(num*num)
```

Is either one better?

lec12append.py

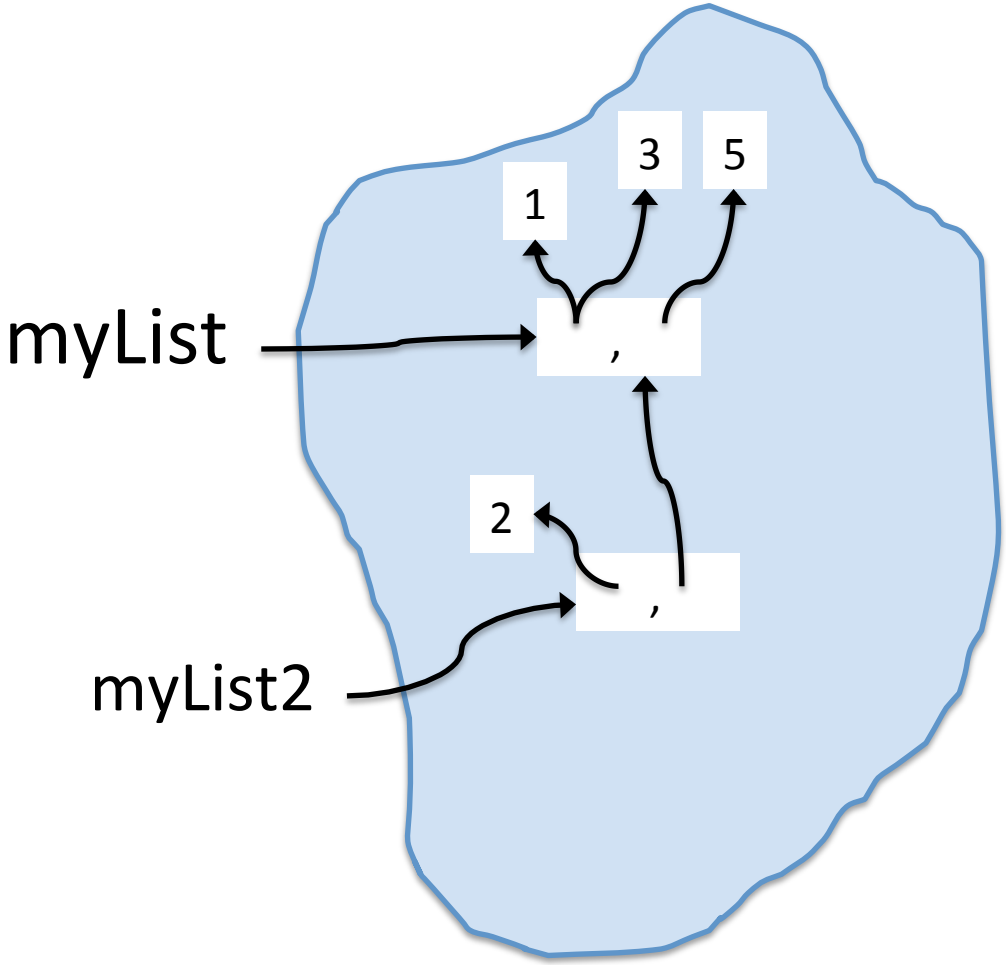
list +

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



Consequences of list mutability

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

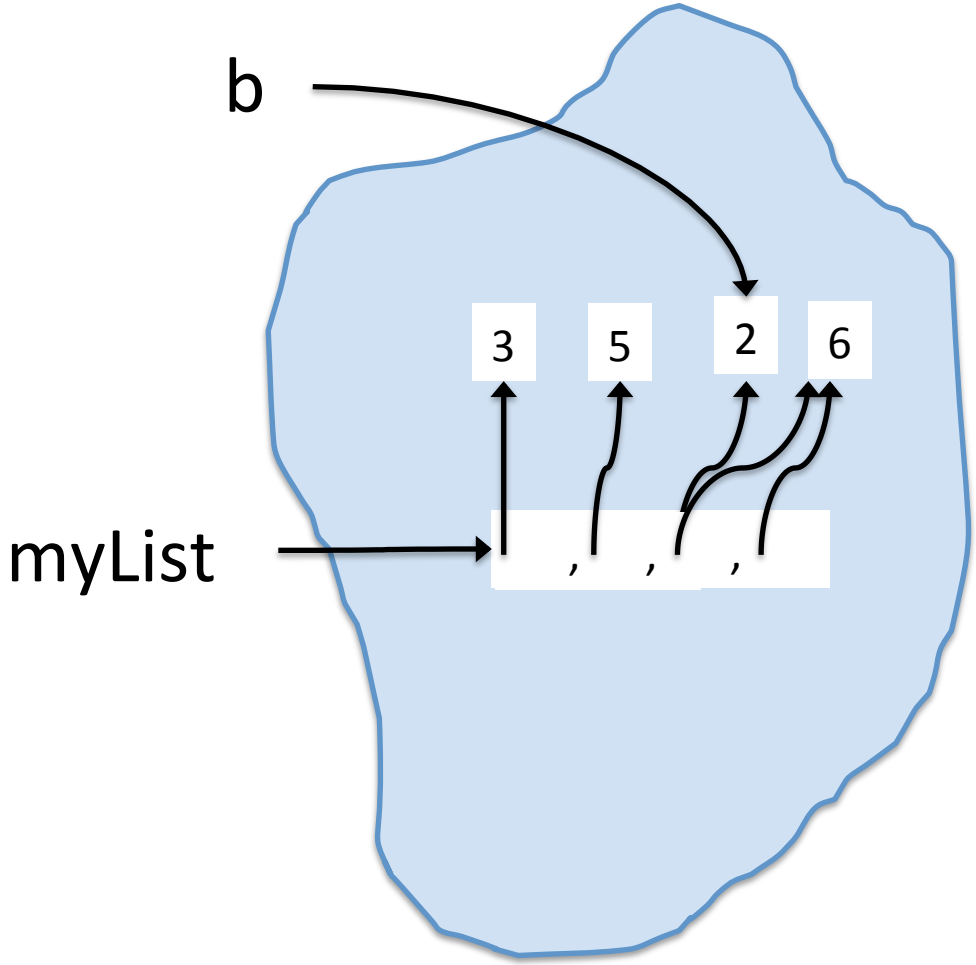


Important when we pass lists as arguments to functions! (next lecture)

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:

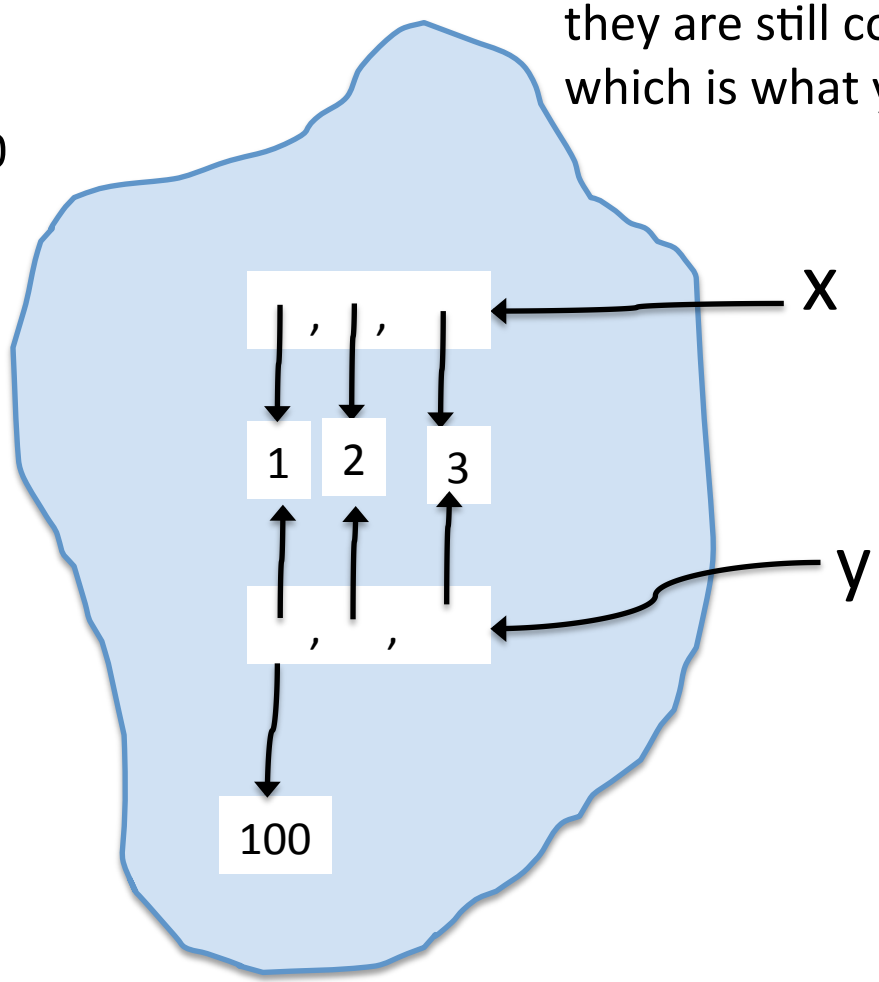
```
>>> x = [1,2,3]
>>> y = x
>>> x is y
True
>>> x[1] = 100
>>> y[1]
?
```

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?

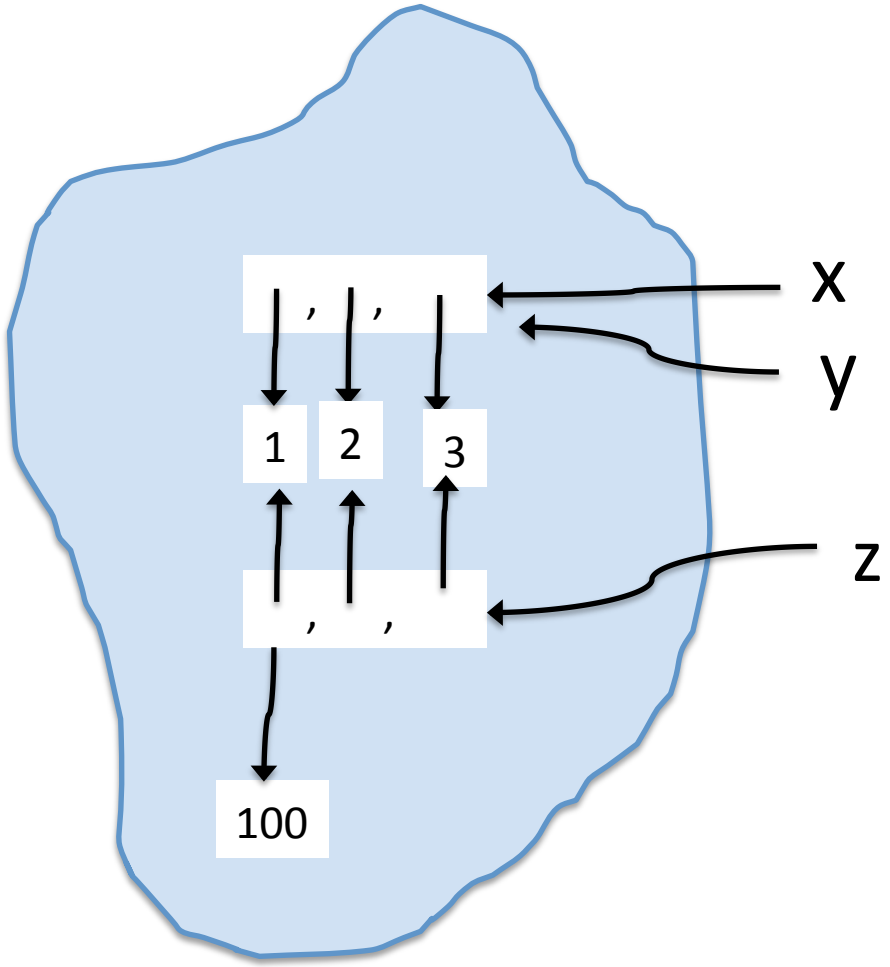
Often, we want to avoid aliasing. So, given a list, can we easily make a copy? YES!

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



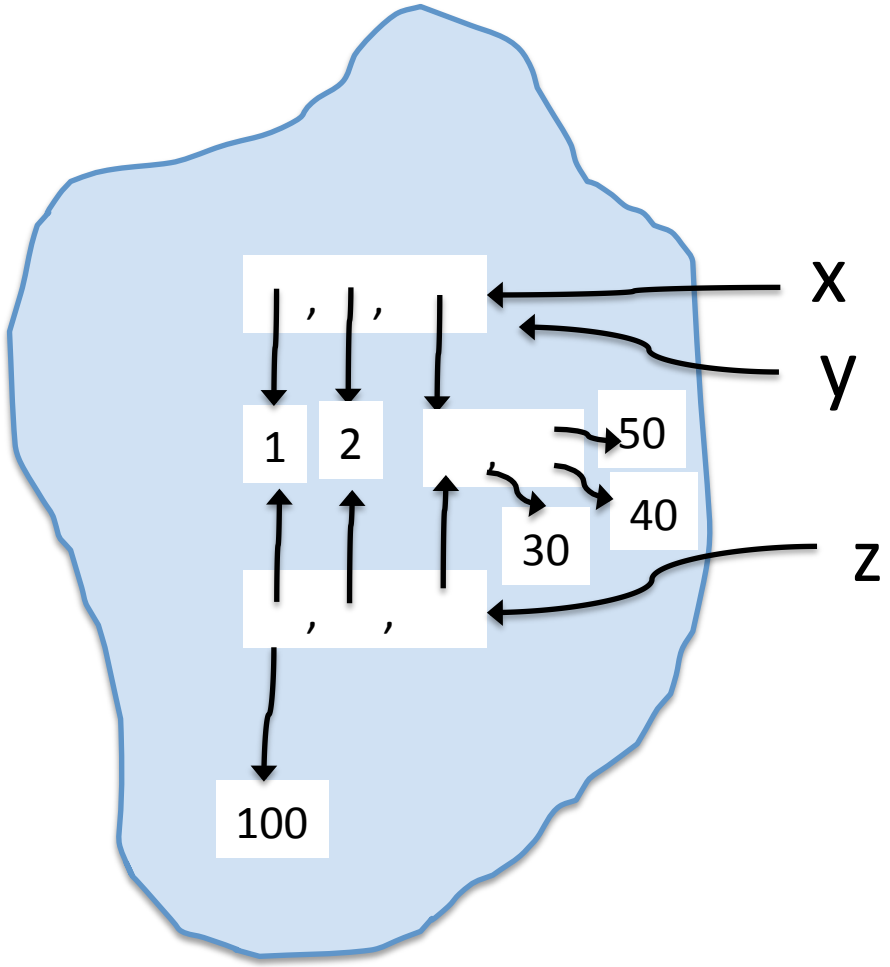
Objects, equality, and identity

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
???
```



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

Next Time

Finish Chapter 10

- aliasing
- lists as arguments to functions

Part of a Ch 19 (“The Goodies”) – list comprehensions

Friday/Monday

Last topic before exam

- recursion (back in Ch 5)