CS1 Lecture 13 Feb. 15, 2017

- HW4 available 11:30 today, due Wed. 2/22, 9 am
- Exam 1, Thursday evening, 2/23, 6:30-8:00pm
- Advice not all may agree with: resist the urge to continually search for better explanations of the material, answers to problems, etc. The text and Python and careful hard work and practice can be enough. This stuff is *not easy*. You might need to read, think, re-read, think, re-read, think, and practice, practice, practice. I see some people "thrashing" – repeatedly searching (often the Internet) for the perfect information source that will provide the "aha" moment. My sense is that the "aha" moment often comes simply after much thought and practice, and with one or two reasonable, dependable very well studied sources. Learn everything in *this* textbook; it's enough!

Last time

Continued Ch 10

- **for** loops on lists
- **Range**s
- for -> while conversion
- more mutability examples and diagrams

Today

Discussion section 4 anagrams example

Finish Ch 10

- more on mutability, including lists as function arguments
- == vs. is, and list copies
- del
- more examples

Discussion section 4 example

- What if we wanted to find the largest set of anagrams?
  - simple direct approach

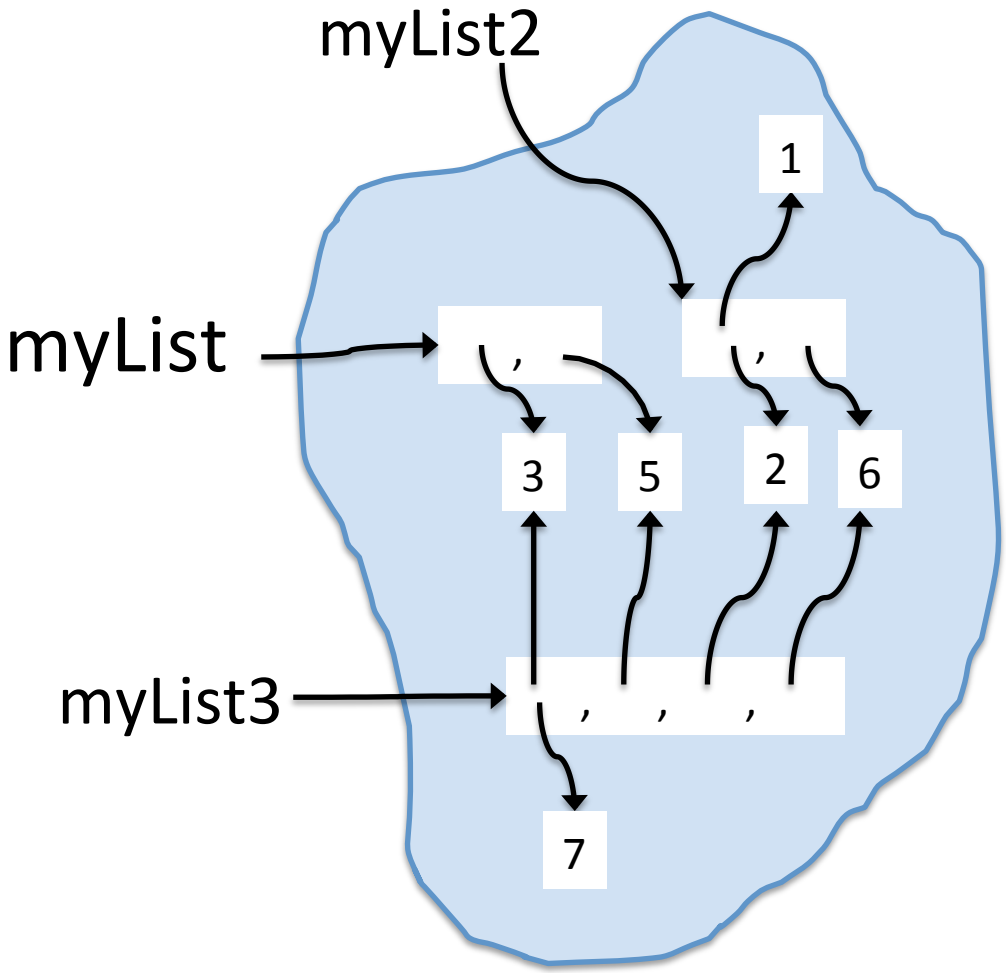
```
biggestAnagramList = []
for word in wordList:
    anagramList = findAnagramsOf(word, wordList)
    if len(anagramList) > len(biggestAnagramList):
        biggestAnagramList = anagramList
```

  - Works okay for a couple thousand words (words5.txt) but far too slow for 100+K list like wordsMany.txt
    - (with doc cam … discussed basic idea for faster approach that we'll look at when we talk more about sorting and run-time analysis in a few week. Idea: associate a "key" with each word, the sorted version of that word. E.g. ["tar", "art"], ["least", "aelst"]. Sort this list of pairs by those "keys". Now all anagrams neighbors in this sorted list and largest set can be found via one simple scan through it. […. ["least", "aelst"], ["stale", "ealst"], ..., ["art", "art"], ["rat", "art"], ["tar", "art"] ...]

biggestAnagramSet.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
?
```

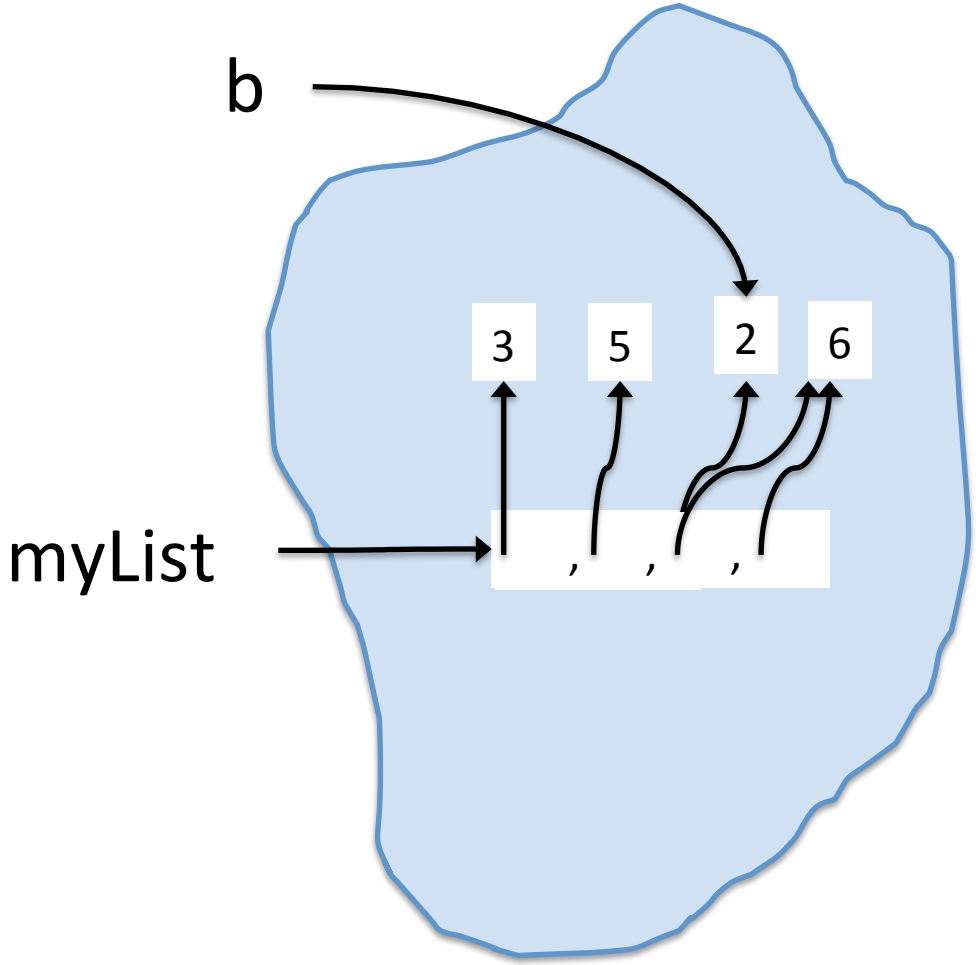


Important to remember: + creates a new list object

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:

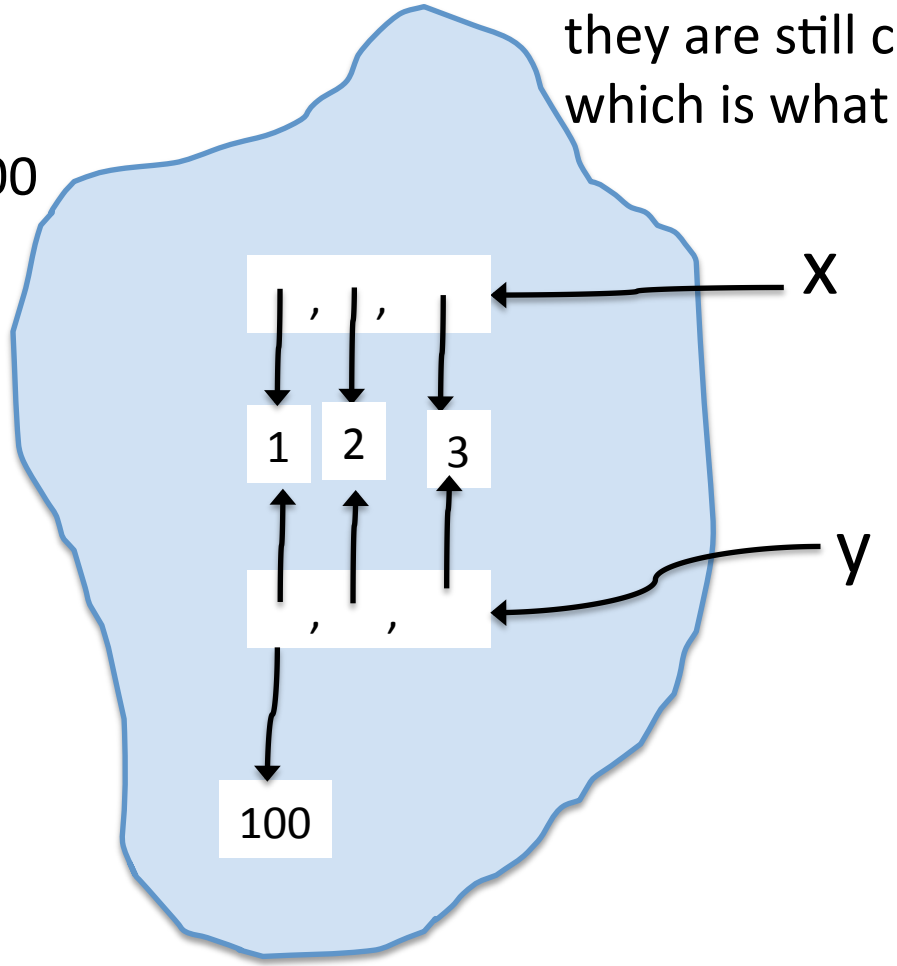
```
>>> x = [1,2,3]
>>> y = x
>>> x is y
True
>>> x[1] = 100
>>> y[1]
?
```

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 consider equal, though,
which is what you usually care about



Objects, equality, and identity

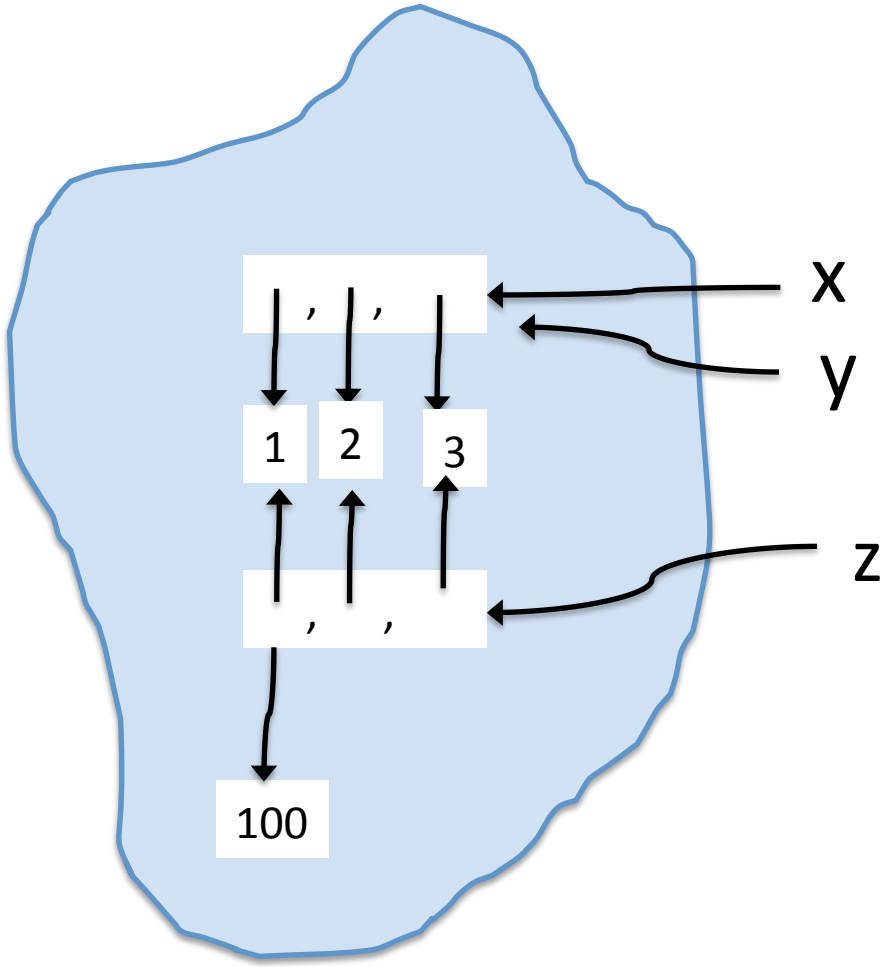
Often, we want to avoid aliasing. So, given a list, can we easily make a copy? YES!

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



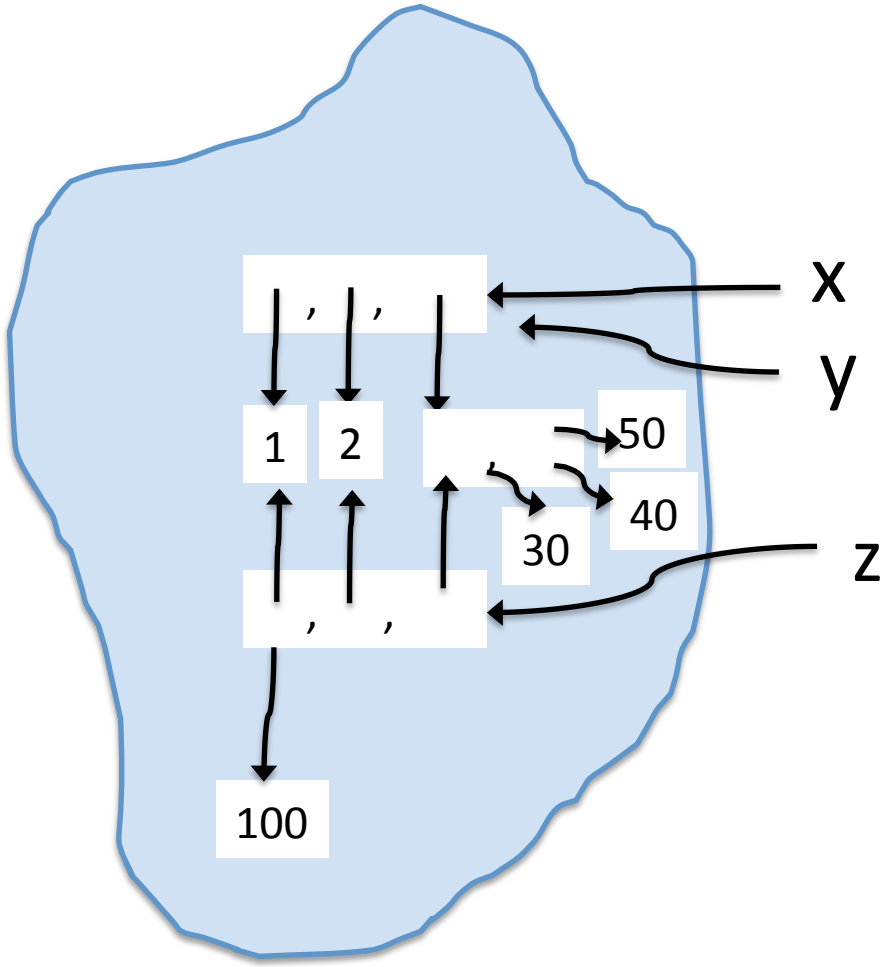
Objects, equality, and identity

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

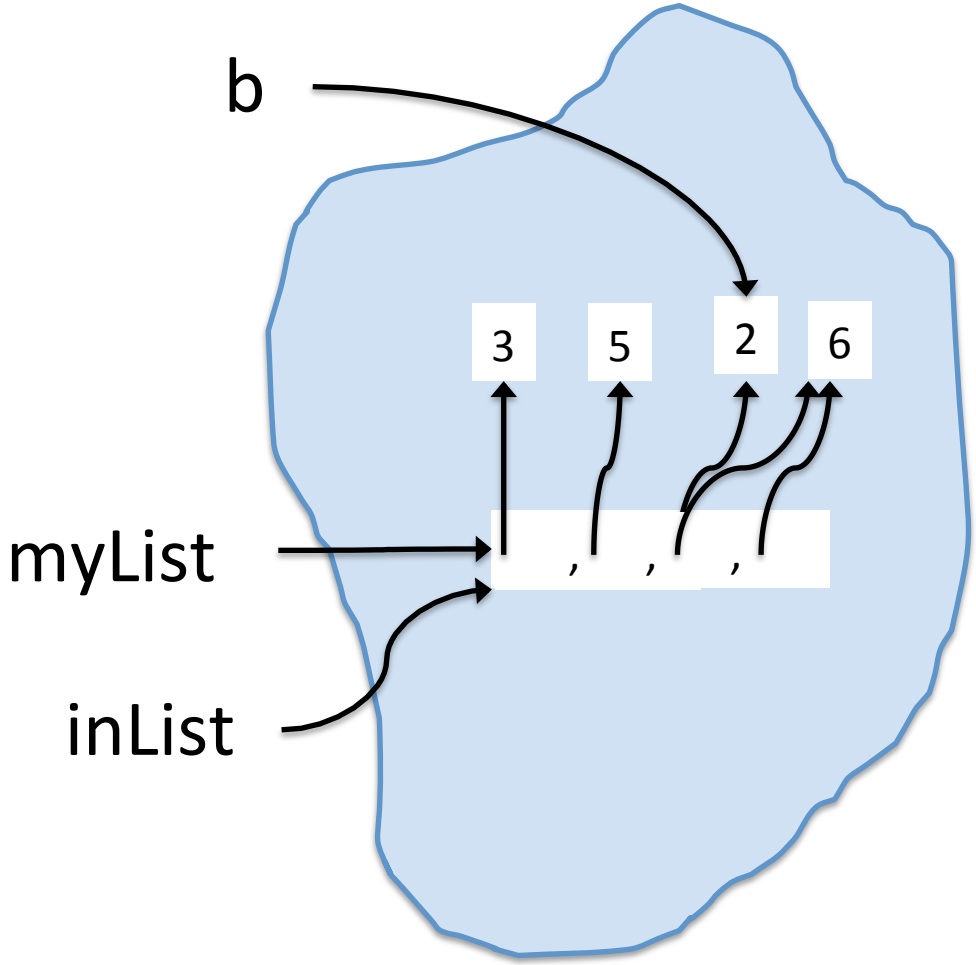
```
>>> x = [1, 2, [30, 40]]
>>> y = x
>>> z = x[:]

>>> 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)

Mutability and arguments to functions

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



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

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.

Consider two examples from exercises at end of Ch 10.

```
# 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

Example: printLetterCounts

Lists make it easy to generalize the printVowelInformation(inputString) function of HW2.

How would you implement printLetterCounts(inputString, letters) that prints the number of occurrences in inputString of each letter in letters?

```
>>> printLetterCounts("This is a sentence containing a variety of letters",
        "aeiouy")
'This is a sentence containing a variety of letters' has:
    4 'a's
    6 'e's
    5 'i's
    2 'o's
    0 'u's
    1 'y's
    and 32 other letters
```

See lec13letterCountsStart.py. Try to complete the function for next time.

Next Time

Last topic before exam

- return to end of Ch 5 - recursion