CS1 Lecture 11 Feb. 10, 2017

- HW3 due Monday morning, 9:00am
  - for #1 I don't care if you use 1, 2, or 3 loops. Most important is clear correct code
  - for #3, make sure all legal situations are handled. Think carefully about how to test.
- Make sure to confirm ICON submissions
  - https://community.canvaslms.com/docs/DOC-3117
- Third survey due tonight!
- TA office hours that were on Monday have been moved to Tuesday and Friday
- HW1 solutions have been posted

Programming advice

Be careful with variable names:

- Don't use ..index.. when it's bound to a value other an index!
- Don't change type of thing variable is bound to – use a different variable!

```
cost1 = 23.0
cost2 = 143.
for index1 in string1:
    index2 = 0
    while index2 < len(string2):
        if string1[index1] == string2[index2]:
            cost1 = "The cost is:" + str(cost1)
        index2 = index2 + 1
…
…
if (cost1 < cost2):
    print("Option 1 is the better one!")
```

HW3 Q1

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

```
# find first and third biggest
    # go through string char by char updating values for
    # three simple variables:
    #    biggest, secondBiggest, and thirdBiggest

# find most common
    # presume you have a function howMany(c, s) that
    # returns the number of times c occurs in s
    # 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 first and third biggest
    # go through string char by char updating values for
    # three simple variables:
    #    biggest, secondBiggest, and thirdBiggest
```

biggest: ~~?~~ ~~e~~ ~~m~~ ~~s~~ y

secondBiggest: ~~?~~ ~~e~~ ~~m~~ s

thirdBiggest: ~~?~~ ~~e~~ ~~f~~ m

e m s b f y

Last time

Finished Chapter 8: Strings, **for**, string **method**s

Mentioned Ch 9 (has good exercises, some covered in disc. section)

Started Chapter 10: **list**s

Ch 10: **lists**

- **list** is another Python sequence type
- In a string, each item of the sequence is a character
- In a list, each item can be a value of any type! (and can be as long as you want)
- The most basic way to create a **list** is to enclose a comma-separated series of values with brackets:

```
>>> [1, 'a', 2.4]
[1, 'a', 2.4]

>>> myList = [1, 'a', 2.4]
>>> len(myList)
 3
>>> myList[0]
1
```

[] operator and len() function work on both strings and lists

Ch 10: **lists**

I said the items in a list be any type. So, can lists be elements of lists? YES!

```
>>> myList = [1, 2, [‘a’, 3]]
>>> len(myList)
3
>>> myList[2]
[‘a’, 3]
>>> myList[2][1]
3
>>> myList[1][2]
Error
```

we call this a “nested list”

Ch 10: **lists**

A list can have no elements!

```
>>> myList = []
>>> len(myList)
0
>>> myList[0]
Error
```

we call this an "empty list"

Ch 10: list operations

slices, +, * work similarly to how they work on strings

```
>>> myList = [1, 2, 3, 4, 5]
>>> myList[1:3]
[2,3]
>>> myList + myList
[1,2,3,4,5,1,2,3,4,5]
>>> myList = myList + [6]
>>> myList
[1,2,3,4,5,6]
>>> myList = myList + 6
Error
>>> myList = myList + [[6]]
>>> myList
[1,2,3,4,5,6,[6]]
>>> 2 * myList
[1,2,3,4,5,6,[6],1,2,3,4,5,6,[6]]
```

Today

More of Ch 10. Much of it is related to important property of lists:

- lists are *mutable!*

It is very important to understand the consequences of list mutability. It can be confusing if you don't take time to understand it!

Ch 10: traversing lists

Just like we often want to iterate through the characters of a string, we often want to “traverse” lists, doing some computation on each list item in turn. Like they are for stirng, **for** loops are again concise and useful

```
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: examples

- Write a function that takes two lists as input and returns a list of all pairs [i1, i2] where i1 in an item from the first list and i2 is an item from the second list pairs
  - e.g. [1,2] and [3,4,5] ->

    [[1,3], [1,4], [1,5], [2,3], [2,4], [2,5]]

- Write a function that, given a list of zero or more sublists of zero or more numbers, returns a list of numbers in which the ith number is the sum of the numbers in the ith sublist.
  - e.g [[2,3], [23], [1,1,1]] -> [5, 23, 3]

lec11.py

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

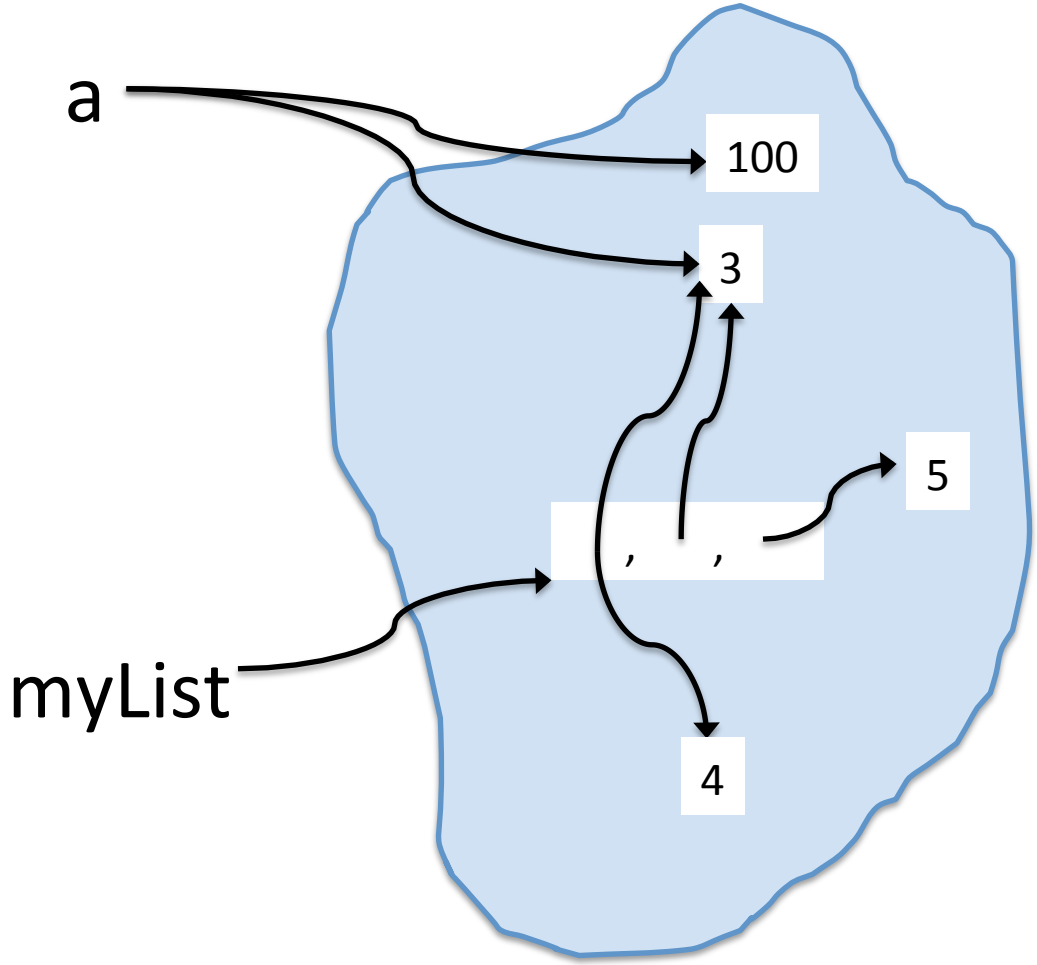
```
>>> myList.append('goodbye')
>>> myList
[1, 53, 'hello', 9, 'goodbye']
```
you can add new items to the end of a list

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



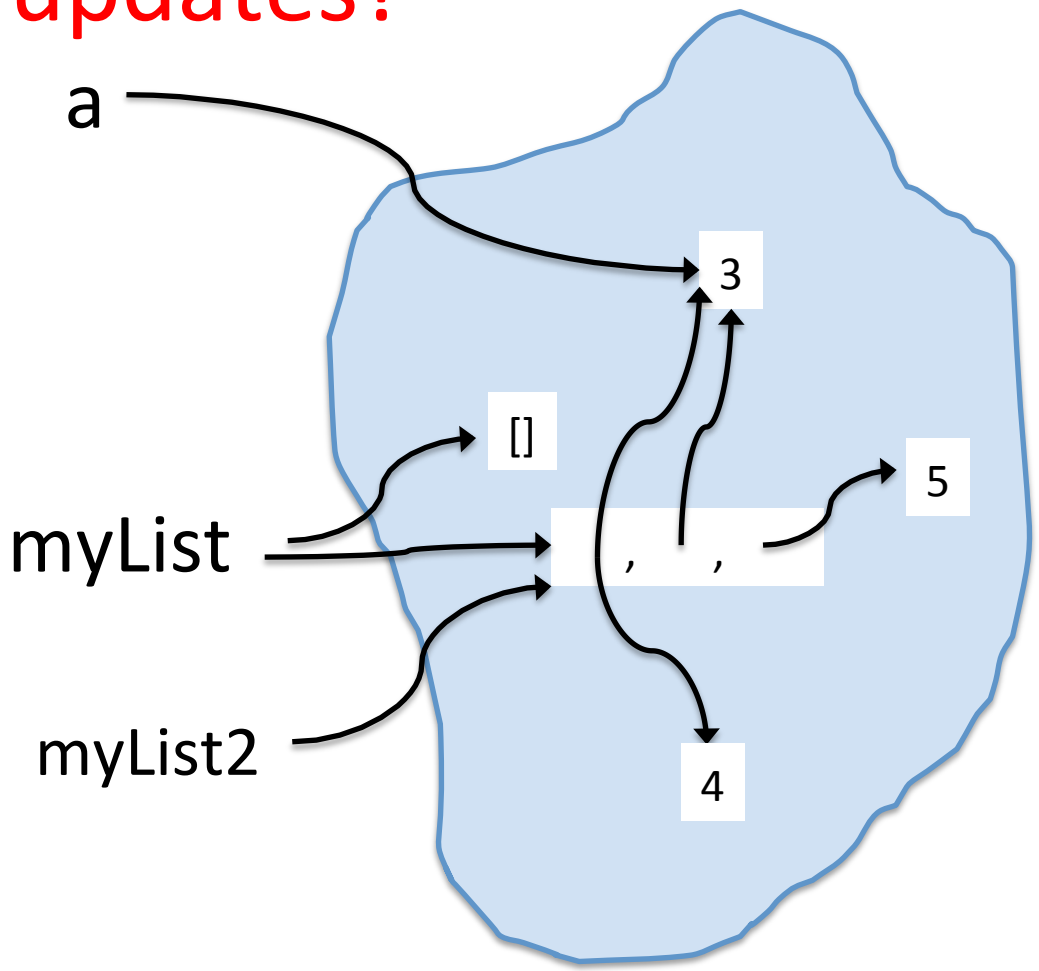
myList[0] = 4 *does not affect a's value!*

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 = []

>>> myList
[]
>>> myList2
???
[4, 3, 5]
```



myList[0] = 4

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

VERY IMPORTANT! CAN BE CONFUSING! We will discuss more next time!

Next Time

Finish Chapter 10

- more on list mutability
- *"aliasing"*
- lists as arguments to functions