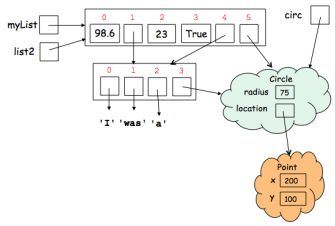
More `for` Loops + Memory Diagrams, & Mutable and Immutable Values



CS111 Computer Programming

Department of Computer Science
Wellesley College

Review: Python `for` loop

Recall that a Python `for` loop performs the loop body for each element of a list.

```
nums = [8,3,10,4,5]

for num in nums:
    print num * 10
```

```
80
30
100
40
50
```

```
for n in nums:
    if (n % 2) == 0:
        print n
```

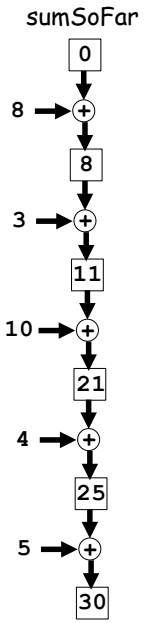
```
8
10
4
```


Accumulating a result with a `for` loop

It's common to use a `for` loop in conjunction with one or more variables ("accumulators") that accumulate results from processing the elements.

E.g., How can we define a sumList function that takes a list of numbers and returns a single number that is their sum?

```
In[3]: sumList([8,3,10,4,5])
Out[3]: 30
```



sumList in Python

```
def sumList(numbers):
    sumSoFar = 0          <- initialize accumulator
    for num in numbers:
        sumSoFar = sumSoFar + num   # or sumSoFar += num
    return sumSoFar
```

update accumulator

return accumulator

for loop: concatAll

	Returns
`concatAll(['To','be','or','not','to','be'])`	`'Tobeornottobe'`
`beatles = ['John','Paul','George','Ringo']`	
`concatAll(beatles)`	`'JohnPaulGeorgeRingo'`
`concatAll([])`	`''`

What should the accumulator do in this case?

```
# Given a list of strings, returns the string that results
# from concatenating them all together
def concatAll(elts):
```


for loop: countOf

```
sentence = 'the cat that ate the mouse liked
            the dog that played with the ball'
```

	Returns
`countOf('the', sentence.split())`	4
`countOf('that', sentence.split())`	2
`countOf('mouse', sentence.split())`	1
`countOf('bunny', sentence.split())`	0
`countOf(3, [1,2,3,4,5,4,3,2,1])`	2

```
# Given a value val and a list elts, returns the
# number of times that val appears in elts
def countOf(val, elts):
```


Looping over the characters of a string

```
# What does this function do?
def mystery(string):
    count = 0
    for char in string:
        if char in 'aeiou':
            count += 1
    return count

mystery('cat')
mystery('mouse')
mystery('xyzzy')
```


Returning early from a loop

When a loop is in a function, **return** can be used to exit the loop early (e.g., before it visits all the elements in a list).

```
def isElementOf(val, elts):
    for e in elts:
        if e == val:
            return True # return (and exit the function)
                        # as soon as val is encountered
    return False  # only get here if val is not in elts

In [1]: sentence = 'the cat that ate the mouse liked
                    the dog that played with the ball'

In [2]: isElementOf('cat', sentence.split())
Out[2]: True # returns as soon as 'cat' is encountered

In [3]: isElementOf('bunny', sentence.split())
Out[3]: False
```

Your turn

	Returns
`containsDigit('The answer is 42')`	`True`
`containsDigit('pi is 3.14159...')`	`True`
`containsDigit('76 trombones')`	`True`
`containsDigit('the cat ate the mouse')`	`False`
`containsDigit('one two three')`	`False`

```
def containsDigit(string):
```


Built-in Python functions

Because many of the operations shown in the previous slides are very common, Python already has built-in functions to perform them.

```
In [13]: sum(range(0,101))
Out[13]: 5050

In [14]: ' '.join(['To','be','or','not','to','be'])
Out[14]: 'To be or not to be'  # .join() is a string method

In [15]: 'the cat ate the mouse'.count('the')
Out[15]: 2       # .count() as a string method

In [16]: ['the','cat','ate','the','mouse'].count('the')
Out[16]: 2       # .count() as a list method

In [17]: 'cat' in sentence.split()
Out[17]: True
```


Lists are Mutable

Lists are **mutable**, meaning that their contents can change over time.

Lists can change in two ways:

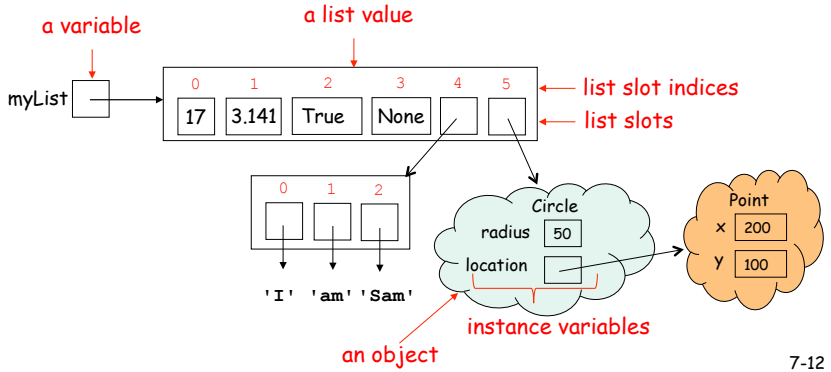
1. The element at a given index can change over time. That is, the slot in a list at a particular index behaves as a **variable**, whose contents can change over time.

2. The length of a list can change over time as new slots are added or removed.

Anatomy of a Memory Diagram

Memory diagrams show the states of variables, lists slots, and object instance variables. They are essential for reasoning about how the state of Python programs can change over time.

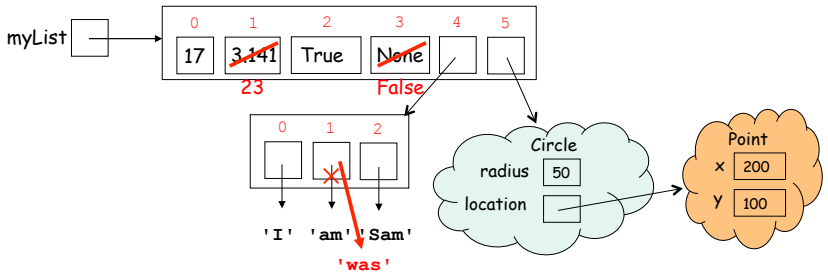
```
myList = [17, 3.141, True, None,
          ['I', 'am', 'Sam'], Circle(50, Point(200, 100))]
```



List Slot Mutability

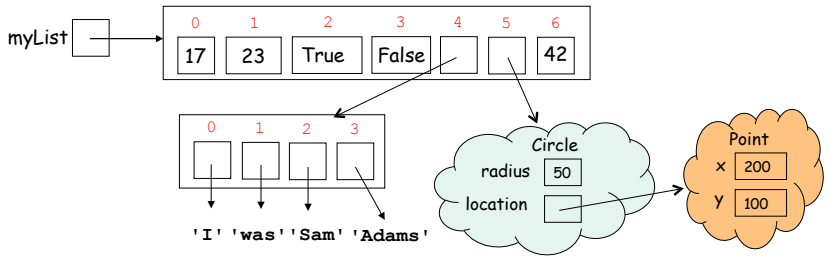
The value in any named or numbered box can change over time.
For example, the values in list slots can be changed by assignment.

```
myList[1] = myList[0] + 6

myList[3] = myList[0] > myList[1]

myList[4][1] = 'was'
```



append: add a new slot to the end of a list

```
myList.append(42)

myList[4].append('Adams')
```



List Mutability

Assigning to a list index:

```
In [67]: numStrings = ['zero', 'one', 'two', 'three', 'four']

In [68]: numStrings[3] = 'THREE'

In [69]: numStrings
Out[69]: ['zero', 'one', 'two', 'THREE', 'four']
```

Adding an element to the end of a list with **append**:

```
In [70]: numStrings.append('five')

In [71]: numStrings
Out[71]: ['zero', 'one', 'two', 'THREE', 'four', 'five']
```


Using `append` to accumulate list results in a `for` loop

Many functions that create and return a list do so by starting with an empty list and using a **`for`** loop to **`append`** elements to this list one at a time.

For example, we can modify the **`sumList`** function to return a list of the partial sums calculated along the way:

```
def partialSums(nums):
    sumSoFar = 0
    partials = []
    for n in nums:
        sumSoFar += n
        partials.append(sumSoFar)
    return partials
```

```
partialSums([8,3,10,4,5])
returns [8,11,21,25,30]
```

Filtering Pattern

A common way to produce a new list is to filter an existing list, keeping only those elements that satisfy a certain predicate. This is called the **filtering pattern**.

```
# Takes a list of numbers and returns a new list of all
# numbers in the input list that are divisible by 2
def filterEvens(nums):
    result = []
    for n in nums:
        if n%2==0:
            result.append(n)
    return result
```

```
filterEvens([8,3,10,4,5]) returns [8,10,4]
filterEvens([8,2,10,4,6]) returns [8,2,10,4,6]
filterEvens([7,3,11,3,5]) returns []
```


Mapping Pattern

Another common way to produce a new list is simply by performing an operation on every element in a given list. This is called the **mapping pattern**.

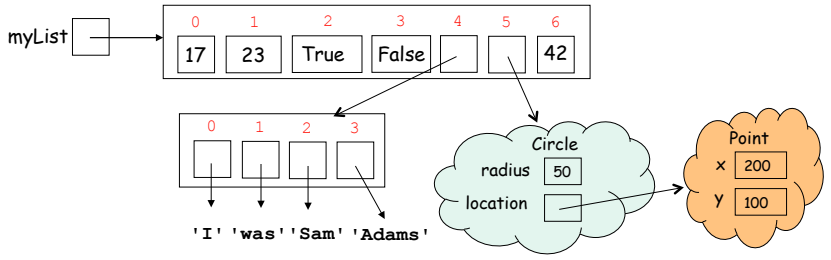
```
# Takes a list of numbers and returns a new list in
# which each element is twice the corresponding element
# in the input list
def mapDouble(nums):
    result = []
    for n in nums:
        result.append(2*n)
    return result
```

```
mapDouble([8,3,10,5,4]) returns [16,6,20,10,8]
mapDouble([17,42,6]) returns [34,84,12]
mapDouble([]) returns []
```


More mutation operation on lists

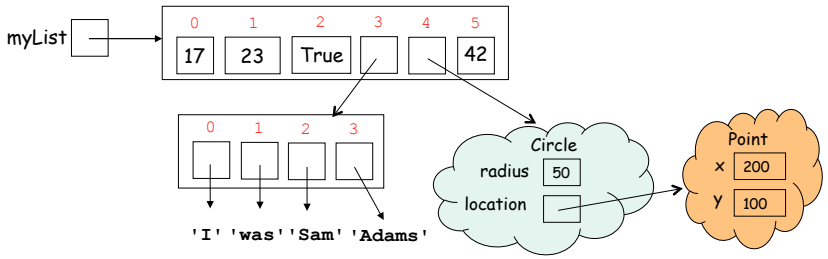
pop: remove slot at an index and return its value

```
myList.pop(3)
```



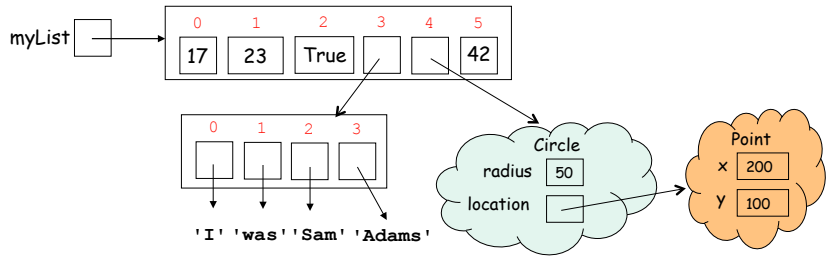
pop: remove slot at an index and return its value

```
myList.pop(3) → False  # Indices of slots after 3 are decremented
```



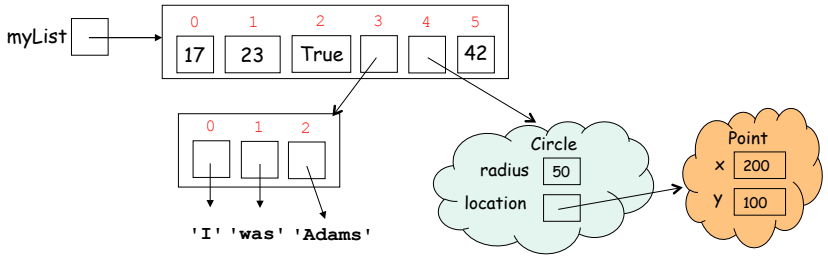
pop: remove slot at an index and return its value

```
myList.pop(3) → False  # Indices of slots after 3 are decremented
myList[3].pop(2)
```



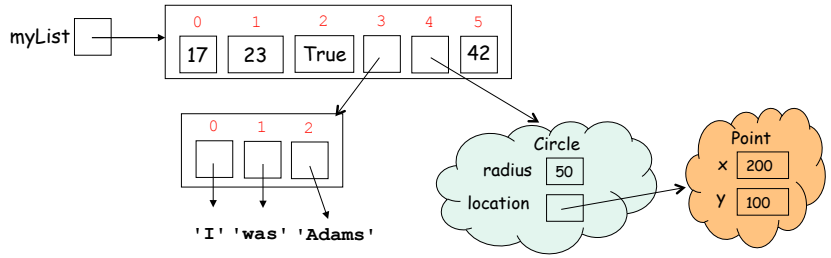
pop: remove slot at an index and return its value

```
myList.pop(3) → False  # Indices of slots after 3 are decremented
myList[3].pop(2) → 'Sam' # Index of previous slot 3 is decremented
```



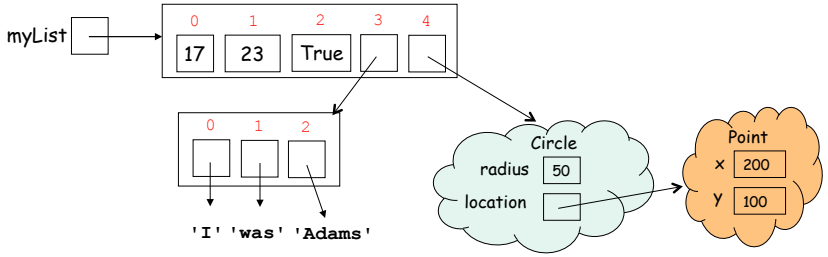
pop: remove slot at an index and return its value

```
myList.pop(3) → False  # Indices of slots after 3 are decremented
myList[3].pop(2) → 'Sam' # Index of previous slot 3 is decremented
myList.pop()
```



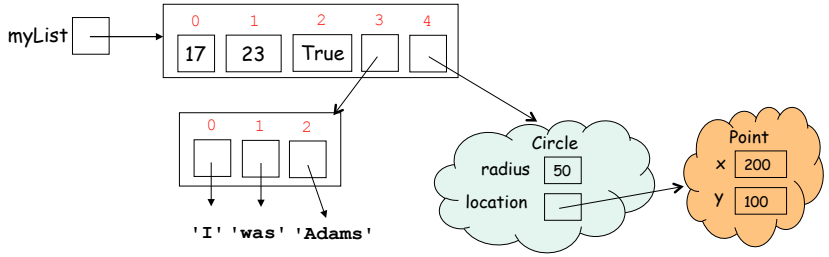
pop: remove slot at an index and return its value

```
myList.pop(3) → False   # Indices of slots after 3 are decremented
myList[3].pop(2) → 'Sam' # Index of previous slot 3 is decremented
myList.pop() → 42   # When no index, last one is assumed
```



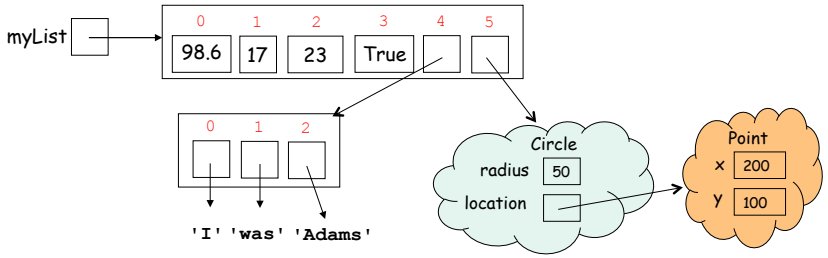
insert: add a slot, add an index

```
myList.insert(0, 98.6)
```



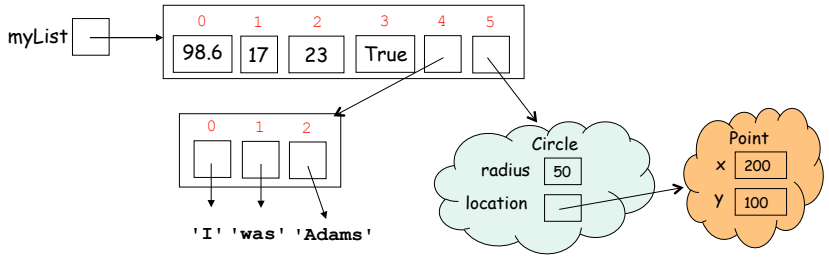
insert: add a slot, add an index

```
myList.insert(0, 98.6)  # Indices of previous slots 0 and above
                        # are incremented
```



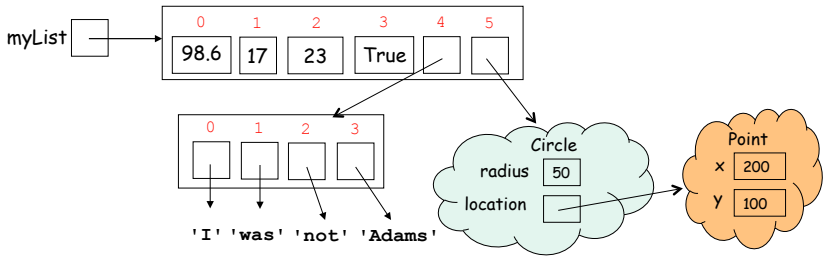
insert: add a slot, add an index

```
myList.insert(0, 98.6)  # Indices of previous slots 0 and above
                        # are incremented

myList[4].insert(2, 'not')
```



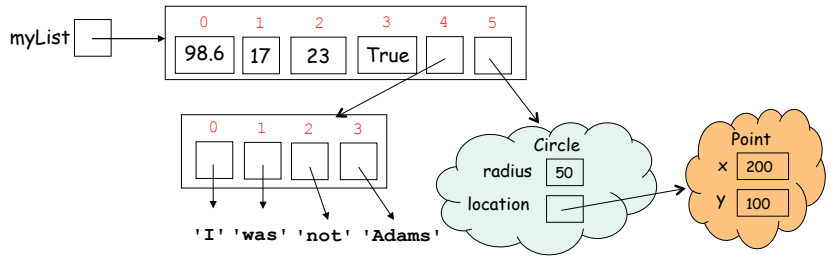
insert: add a slot, add an index

```
myList.insert(0, 98.6)  # Indices of previous slots 0 and above
                        # are incremented

myList[4].insert(2, 'not') # Index of previous slot 2 is incremented
```



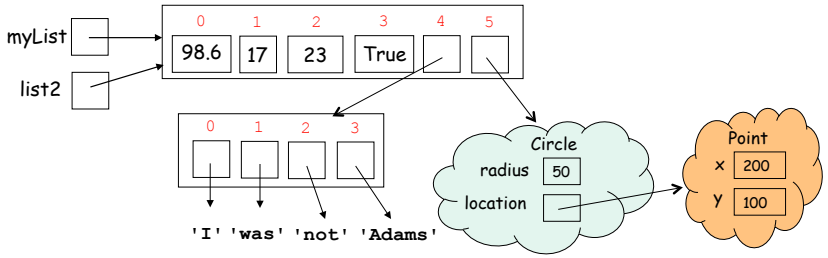
Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
```



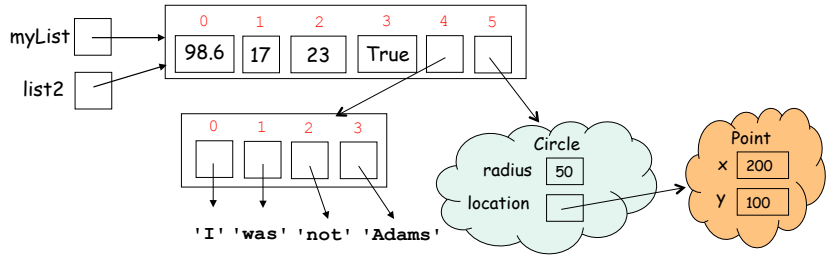
Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
```



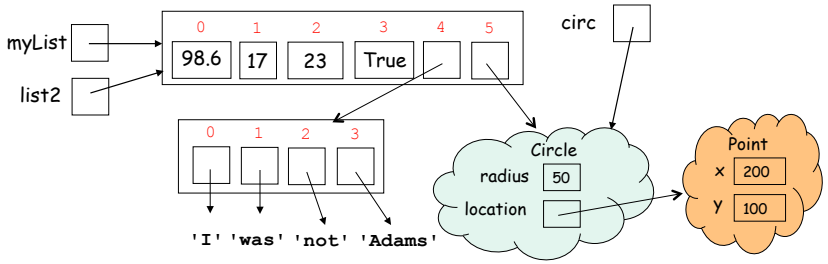
Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
```



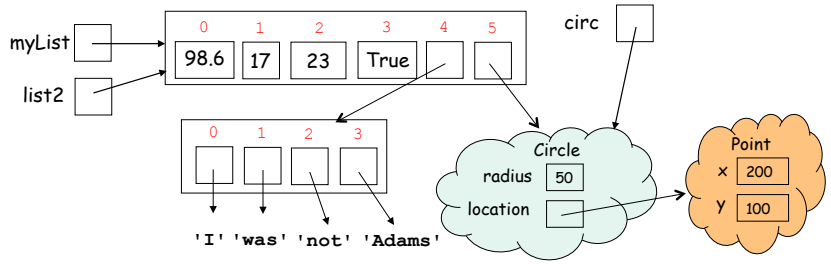
Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
```



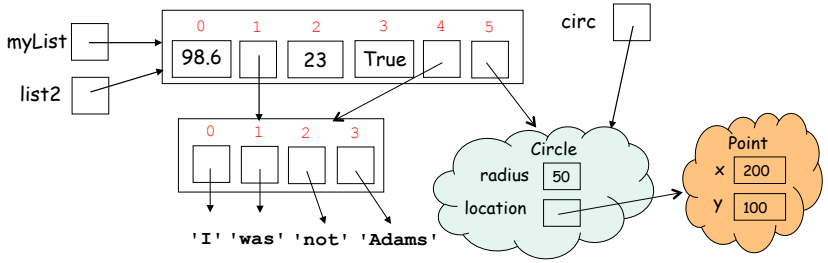
Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
myList[1] = myList[4]
```



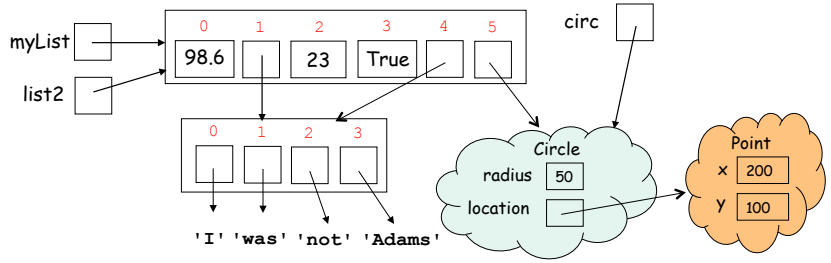
Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
myList[1] = myList[4]
```



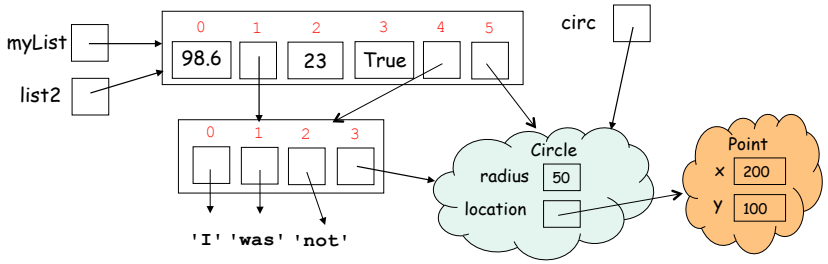
Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
myList[1] = myList[4]
myList[1][3] = circ
```



Aliasing: the very same object can be stored in different variables & slots

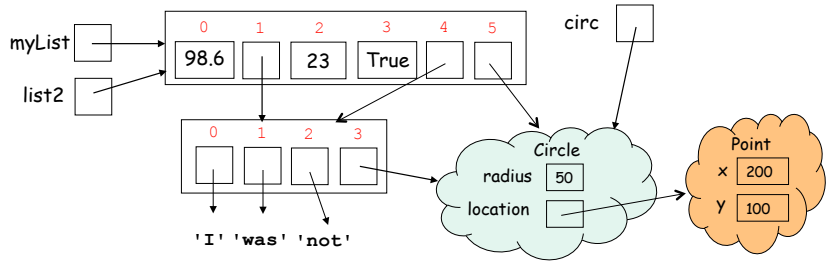
```
list2 = myList
circ = list2[5]
myList[1] = myList[4]
myList[1][3] = circ
```



Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
myList[1] = myList[4]
myList[1][3] = circ
```

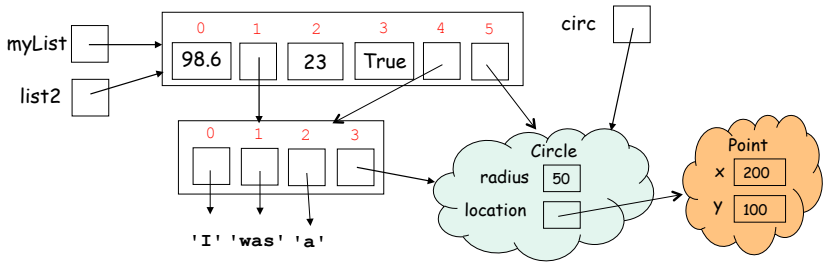
```
myList[1][2] = 'a'
```



Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
myList[1] = myList[4]
myList[1][3] = circ
```

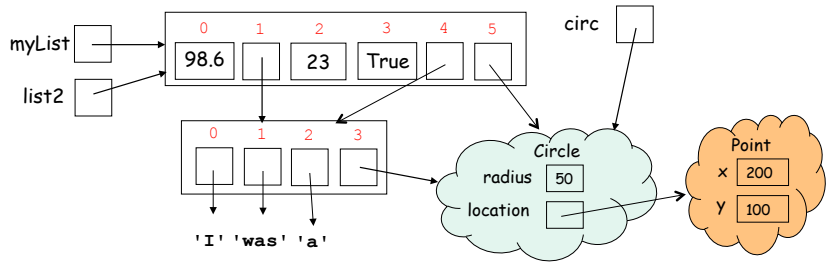
```
myList[1][2] = 'a'
```



Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
myList[1] = myList[4]
myList[1][3] = circ
```

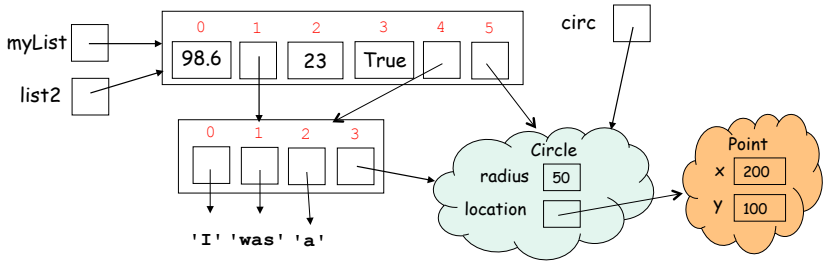
```
myList[1][2] = 'a'
list2[4][2]
```



Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
myList[1] = myList[4]
myList[1][3] = circ
```

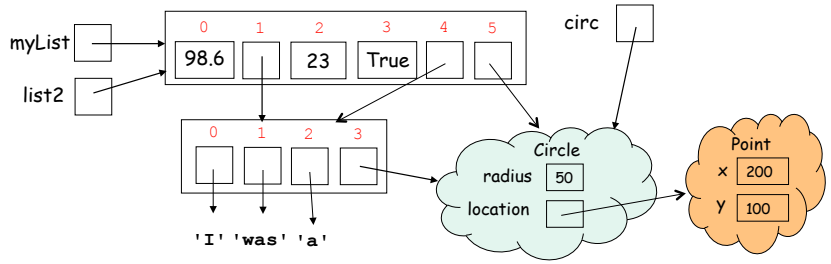
```
myList[1][2] = 'a'
list2[4][2] → 'a'
```



Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
myList[1] = myList[4]
myList[1][3] = circ
```

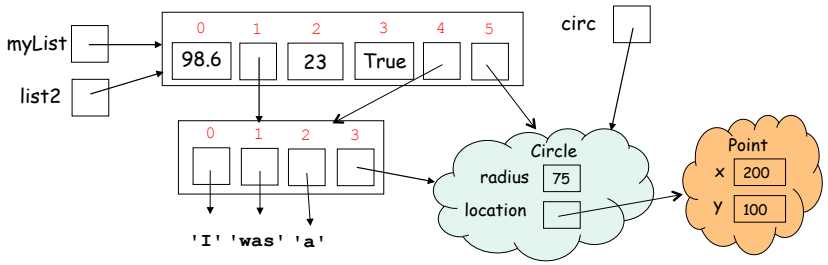
```
myList[1][2] = 'a'
list2[4][2] → 'a'
myList[5].setRadius(75)
```



Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
myList[1] = myList[4]
myList[1][3] = circ
```

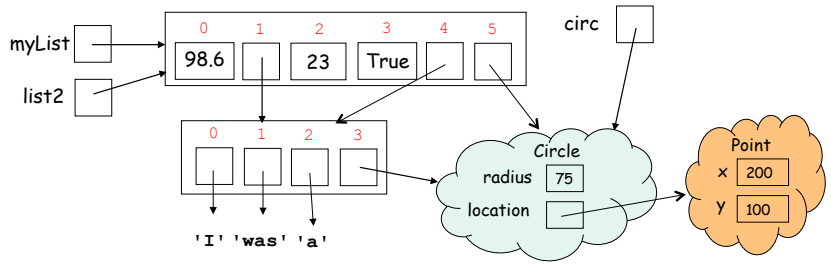
```
myList[1][2] = 'a'
list2[4][2] → 'a'
myList[5].setRadius(75)
```



Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
myList[1] = myList[4]
myList[1][3] = circ
```

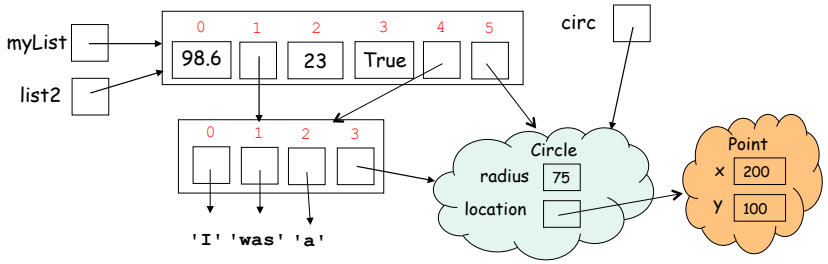
```
myList[1][2] = 'a'
list2[4][2] → 'a'
myList[5].setRadius(75)
circ.getRadius()
```



Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
myList[1] = myList[4]
myList[1][3] = circ
```

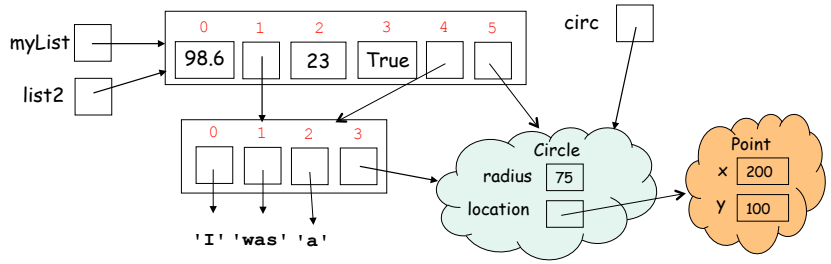
```
myList[1][2] = 'a'
list2[4][2] → 'a'
myList[5].setRadius(75)
circ.getRadius() → 75
```



Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
myList[1] = myList[4]
myList[1][3] = circ
```

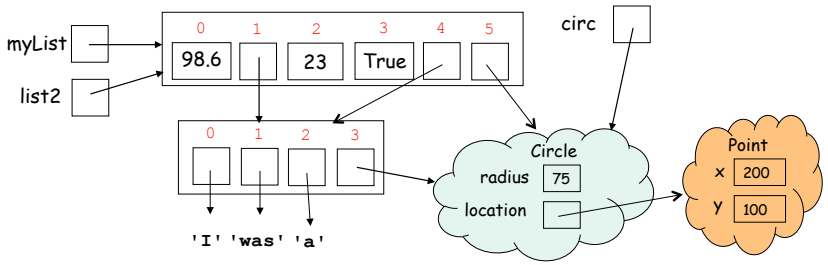
```
myList[1][2] = 'a'
list2[4][2] → 'a'
myList[5].setRadius(75)
circ.getRadius() → 75
list2[1][3].getRadius()
```



Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
myList[1] = myList[4]
myList[1][3] = circ
```

```
myList[1][2] = 'a'
list2[4][2] → 'a'
myList[5].setRadius(75)
circ.getRadius() → 75
list2[1][3].getRadius() → 75
```



Pop Quiz! What is the final value of c[0]?

```
a = [15, 20]
b = [15, 20]
c = [10, a, b]
b[1] = 2*a[0]
c[1][0] = c[0]
c[0] = a[0] + c[1][1] + b[0] + c[2][1]
```

Draw a memory diagram!

Does the answer change if we change the 2nd line from `b = [15, 20]` to `b = a[:]`?

Does the answer change if we change the 2nd line from `b = [15, 20]` to `b = a`?

Tuples

Tuples are **immutable** sequences of values, in contrast with lists, which are **mutable** sequences of values.

Tuples are written as comma-separated values delimited by parentheses

```
(5, 8, 7, 1, 3)  # A homogeneous tuple of five integers
('John', 'Paul', 'George', 'Ringo')  # A homogeneous tuple of four
                                     # strings
(42, 'Hello', False)  # A heterogeneous tuple with three elements
(7, 3)  # A pair is a tuple with two elements
(7, )  # A tuple with one element must use a comma to avoid being
       # confused with a paranthesized expression
( )  # A tuple with 0 values
```


Tuples have all the operations of lists that don't involve mutation

```
In[32]: beatTup = ('John', 'Paul', 'George', 'Ringo')

In[33]: beatTup[2]
Out[33]:'George'

In[34]: beatTup[1:3]
Out[34]: ('Paul', 'George')

In[35]: beatTup.count('Paul')
Out[35]: 1

In[36]: 'Ringo' in beatTup
Out[36]: True

In[37]: beatTup*2 + ('Pete',)
Out[37]: ('John', 'Paul', 'George', 'Ringo', 'John', 'Paul',
          'George', 'Ringo', 'Pete')
```


Mutation operations don't work on tuples

```
In [41]: beatTup[0] = 'Julian'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-40-8e6b1252c619> in <module>()beatTup[0] = 'Julian'
TypeError: 'tuple' object does not support item assignment

In [41]: beatTup.append('Pete')
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-41-e58c65e73cf0> in <module>()----> 1
beatTup.append('Pete')
AttributeError: 'tuple' object has no attribute 'append'

In [42]: beatTup.pop(1)
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-42-67e4d82eec44> in <module>()----> 1 beatTup.pop(1)
AttributeError: 'tuple' object has no attribute 'pop'
```


Strings

A string is effectively a tuple of characters (and so it is **immutable**). E.g., can't do:

```
In [43]: s = 'fan'
In [44]: s[1] = 'u'
TypeError: 'str' object does not support item
assignment
```

A string differs from a tuple in that:

- all of its elements must be characters (e.g., single-character strings)
- it is written as a sequence of characters delimited by single or double quotation marks

String Operations

Of course, all of the tuple operations work for strings as well.

```
In [63]: 'CS' + '1'*3
Out[63]: 'CS111'

In [64]: len('Computer programming'[1:12:2])
Out[64]: 6

In [65]: 'ogra' in 'Computer programming'
Out[65]: True

In [66]: 'Computer programming'.count('m')
Out[66]: 3
```


Type Conversion

The built-in functions `str()`, `list()`, `tuple()` create a new value of the corresponding type.

```
In [64]: word = "Wellesley"
In [65]: list(word)
Out[65]: ['W', 'e', 'l', 'l', 'e', 's', 'l', 'e', 'y']

In [66]: tuple(word)
Out[66]: ('W', 'e', 'l', 'l', 'e', 's', 'l', 'e', 'y')

In [67]: numbers = range(5,15,2)
In [68]: str(numbers)
Out[68]: '[5, 7, 9, 11, 13]'
```