Lists,
Memory Diagrams,
Mutable vs. Immutable Sequences

CS111 Computer Programming

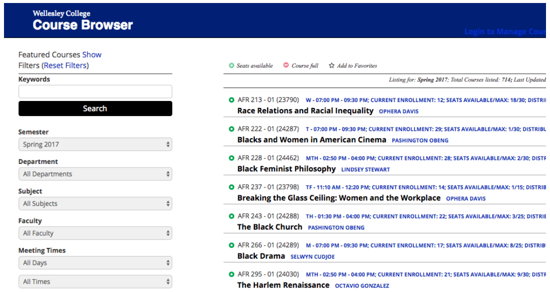
Department of Computer Science
Wellesley College

Why Lists (and other sequences)?

Lists (and other sequences) are useful to represent collections, especially where order matters.

Complete works of Maya Angelou:

- As a single string
- As a list of books, poems, sentences, verses, words, etc.



Wellesley College
Course Browser

Login to Manage Cour

Featured Courses Show
Filters (Reset Filters)

Keywords

Search

Semester: Spring 2017
Department: All Departments
Subject: All Subjects
Faculty: All Faculty
Meeting Times: All Days
All Times

○ Seats available ⊖ Course full ☆ Add to Favorites

*Listing for: **Spring 2017**; Total Courses listed: **714**; Last Updated*

○ AFR 213 - 01 (23790) W - 07:00 PM - 09:30 PM; CURRENT ENROLLMENT: 12; SEATS AVAILABLE/MAX: 18/30; DISTRI
Race Relations and Racial Inequality OPHERA DAVIS

○ AFR 222 - 01 (24287) T - 07:00 PM - 09:30 PM; CURRENT ENROLLMENT: 29; SEATS AVAILABLE/MAX: 1/30; DISTRIBU
Blacks and Women in American Cinema PASHINGTON OBENG

○ AFR 228 - 01 (24462) MTH - 02:50 PM - 04:00 PM; CURRENT ENROLLMENT: 28; SEATS AVAILABLE/MAX: 2/30; DISTR
Black Feminist Philosophy LINDSEY STEWART

○ AFR 237 - 01 (23798) TF - 11:10 AM - 12:20 PM; CURRENT ENROLLMENT: 14; SEATS AVAILABLE/MAX: 1/15; DISTRIB
Breaking the Glass Ceiling: Women and the Workplace OPHERA DAVIS

○ AFR 243 - 01 (24288) TH - 01:30 PM - 04:00 PM; CURRENT ENROLLMENT: 22; SEATS AVAILABLE/MAX: 3/25; DISTRIE
The Black Church PASHINGTON OBENG

○ AFR 266 - 01 (24289) M - 07:00 PM - 09:30 PM; CURRENT ENROLLMENT: 17; SEATS AVAILABLE/MAX: 8/25; DISTRIB
Black Drama SELWYN CUDJOE

○ AFR 295 - 01 (24030) MTH - 02:50 PM - 04:00 PM; CURRENT ENROLLMENT: 21; SEATS AVAILABLE/MAX: 9/30; DISTR
The Harlem Renaissance OCTAVIO GONZALEZ

course information
for all Wellesley courses

list of all registered US voters

Homogenous and nested lists

Lists in which all elements have the same type are called **homogeneous**.

Most of the lists we'll use will be homogeneous.

```
# List of primes less than 20
[2, 3, 5, 7, 11, 13, 17, 19]
```

Lists can also contain other lists as elements! These are **nested lists**.

```
# List of string lists
[['fox', 'raccoon'], ['duck', 'raven', 'gosling'], [], ['turkey']]
```

In this lecture, we’ll use heterogenous lists for pedagogical purposes.

Heterogeneous lists

Python also allows **heterogeneous** lists in which elements can have different types.

```
[17, True, ‘R-Rated', None, [13, False, ‘PG-13']]
```

In general, you should avoid heterogeneous lists unless you have a good reason to use them (they make programs harder to reason about).

In a few weeks, we will learn about another data type (dictionaries) that are better at storing heterogeneous data.

Lists: glue for many values

```
# Lists returned from built-in functions and methods
odds   = list(range(1,10,2))  # [1,3,5,7,9]
lyrics  = 'call me on my cell'.split() # splits by spaces
                             # ['call', 'me', 'on', 'my', 'cell']
letters = list('happy')  # ['h', 'a', 'p', 'p', 'y']


# Literal list definitions
primes  = [2, 3, 5, 7, 11, 13, 17, 19]
bools   = [1<2, 1==2, 1>2]
houses  = ['Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin']
strings = ['ab' + 'cd', 'ma'*4]
counts  = [1, 2, 3] + [4, 5]
animalLists = [['fox', 'raccoon'],
                ['duck', 'raven', 'gosling'], [], ['turkey']]

# A heterogeneous list
stuff = [17, True, 'foo', None, [42, False, 'bar']]

# An empty list
empty = []
```

List sequence operations (review)

Operation	Result
`x in seq`	True if an item of seq is equal to x
`x not in seq`	False if an item of seq is equal to x
`seq1 + seq2`	The concatenation of seq1 and seq2
`seq*n, n*seq`	n copies of seq concatenated
`seq[i]`	i'th item of seq, where origin is 0
`seq[i:j]`	slice of seq from i to j
`seq[i:j:k]`	slice of seq from i to j with step k
`len(seq)`	length of seq
`min(seq)`	smallest item of seq
`max(seq)`	largest item of seq

List indexing and slicing (review)

```
In[1]:   houses  = ['Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin']

In[2]:   houses[0]      # List indexing
Out[2]: 'Gryffindor'

In[3]:   houses[3]
Out[3]: 'Slytherin'

In[4]:   houses[4]
---------------------------------------------
IndexError  Traceback (most recent call last)
<ipython-input-4-834fac18ce76> in <module>()
----> 1 houses[4]
IndexError: list index out of range
```

Indexing: get one element from the given position (index) in the list.

List indexing and slicing (review)

```
In[5]:   houses[-3]
Out[5]:  'Hufflepuff'

In[6]:   houses[1:3] # List slicing
Out[6]:  ['Hufflepuff', 'Ravenclaw']

In[7]:   houses[2:]
Out[7]:  ['Ravenclaw', 'Slytherin']

In[8]:   houses[:2]
Out[8]:  ['Gryffindor', 'Hufflepuff']
```

Negative indexing: negative indices index from the end of the list.

Slicing: get a new list of all list elements at indices in the given ***range***.

Nested list indexing (is not special!)

```
In[1]:   animalLists = [['fox', 'raccoon'],
                        ['duck', 'raven', 'gosling'],
                        [],
                        ['turkey']]
In[2]:   animalLists[0][1]
Out[2]:  'raccoon'

In[3]:   mammals = animalLists[0]

In[4]:   mammals
Out[4]:  ['fox', 'raccoon']

In[5]:   mammals[1]
Out[5]:  'raccoon'
```

List of lists.

Nested list indexing is not special! It is just indexing an already indexed sequence.

Practice!

```
animalLists = [['fox', 'raccoon'],
               ['duck', 'raven', 'gosling'],
               [],
               ['turkey']]
```

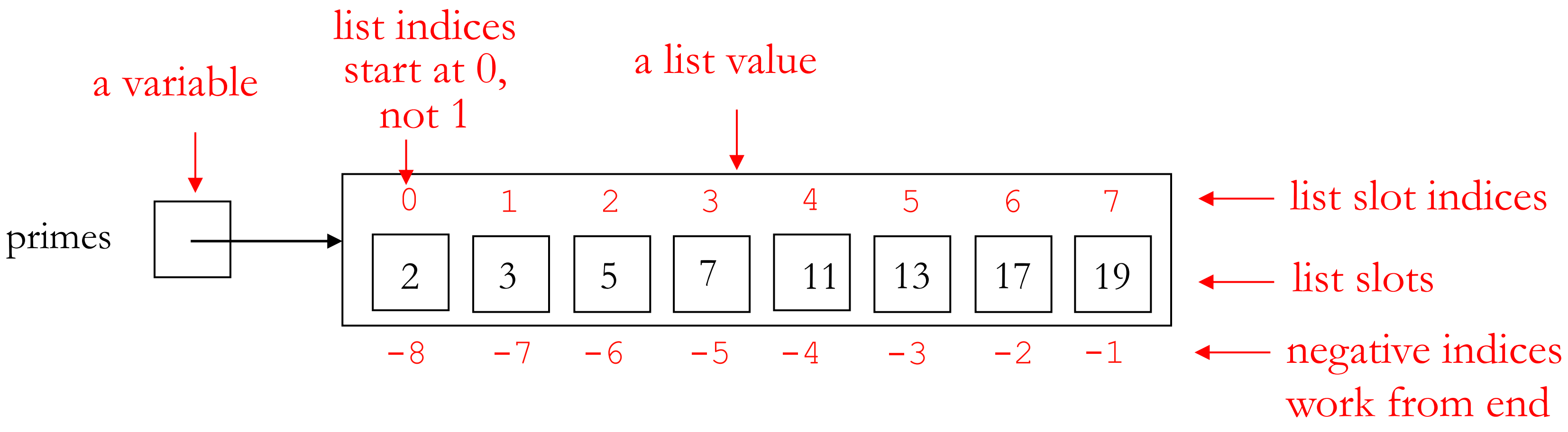
Write a 1-line Python expression to get `'raven'` from **`animalLists.`**

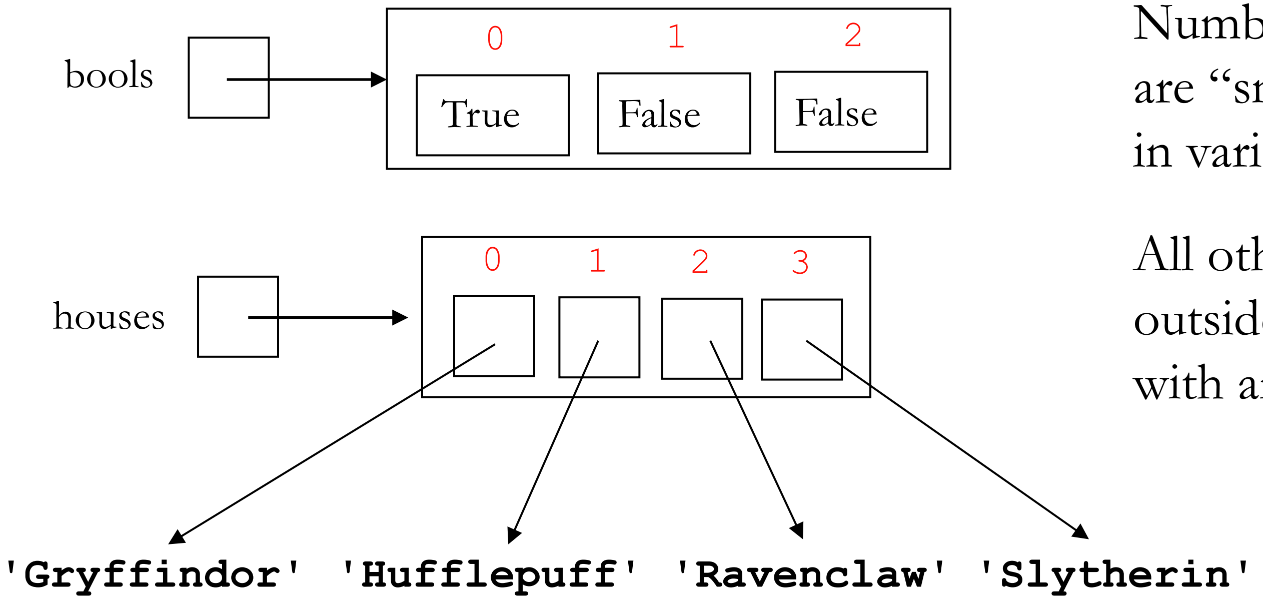
Write a 1-line Python expression to get `'turkey'` from **`animalLists.`**

Challenge: write two new expressions that also get `'raven'` and `'turkey'` using different indices than before.

How to represent list values: Memory Diagrams [1]

Big IDEA # 4: Models

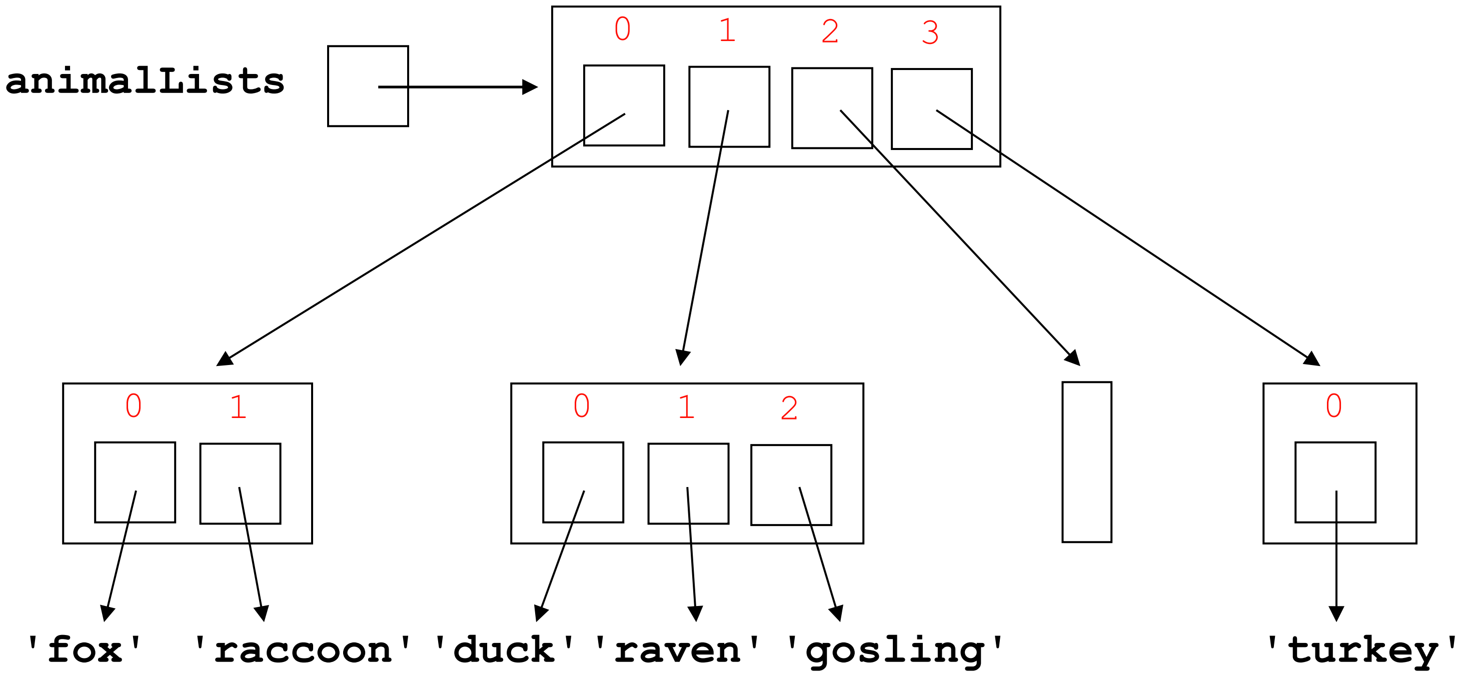




Numbers, booleans, and `None` are “small enough” to fit directly in variables and list slots.

All other values are drawn outside the variable/list slot, with an arrow pointing to them.

How to represent nested lists: Memory Diagrams [2]



```
animalLists = [['fox', 'raccoon'],
               ['duck', 'raven', 'gosling'],
               [],
               ['turkey']]
```

Lists are mutable/changeable

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

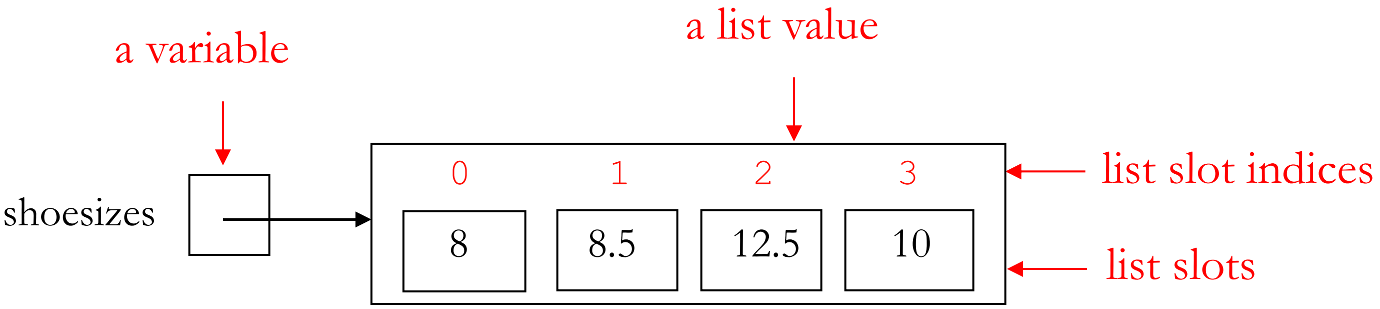
Lists can **change** in two ways:

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.

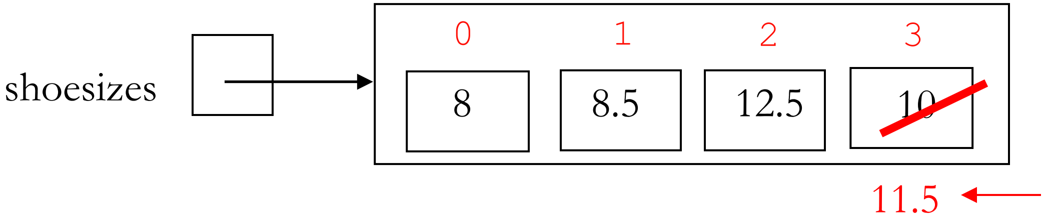
2. The length of a list can change over time as new slots are added or removed.

List slot mutability example

```
shoesizes = [8, 8.5, 12.5, 10]
```



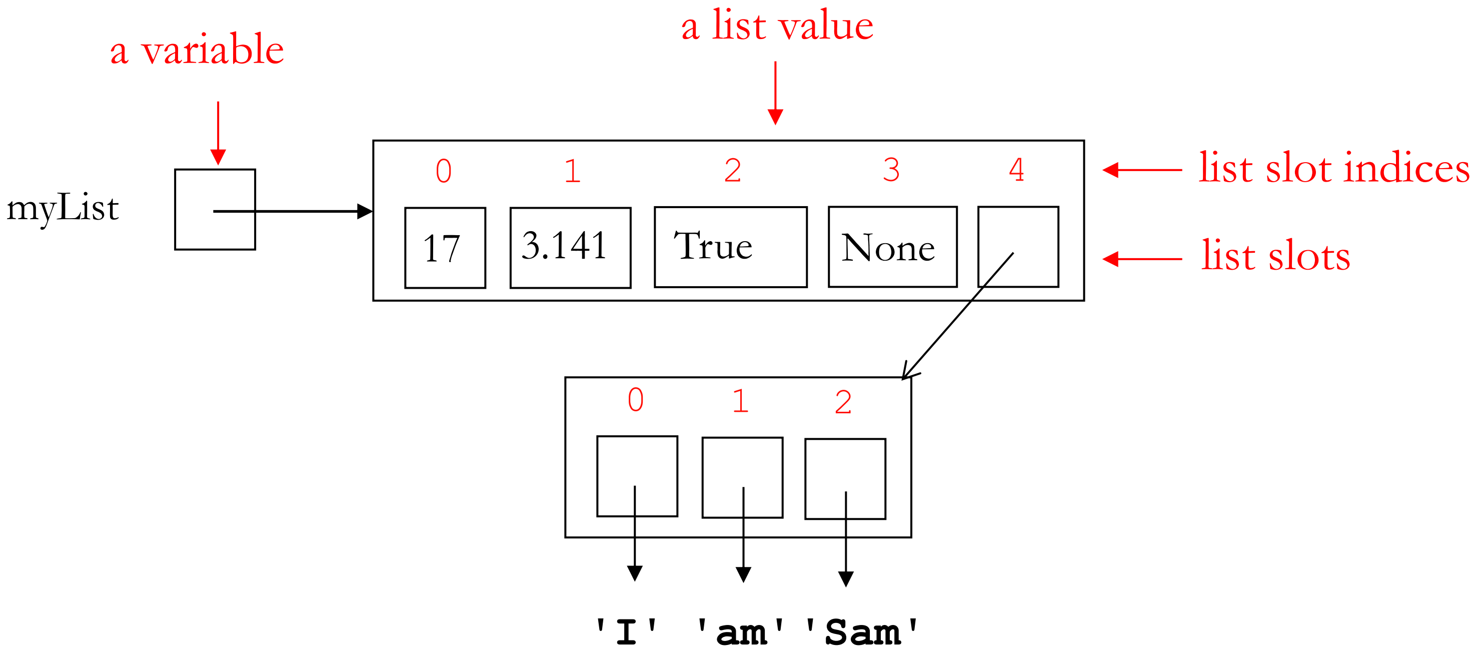
```
shoesizes[3] = 11.5
```



after this change, there is no "memory" that shoesizes[3] was previously 10

List slot mutability larger example [1]

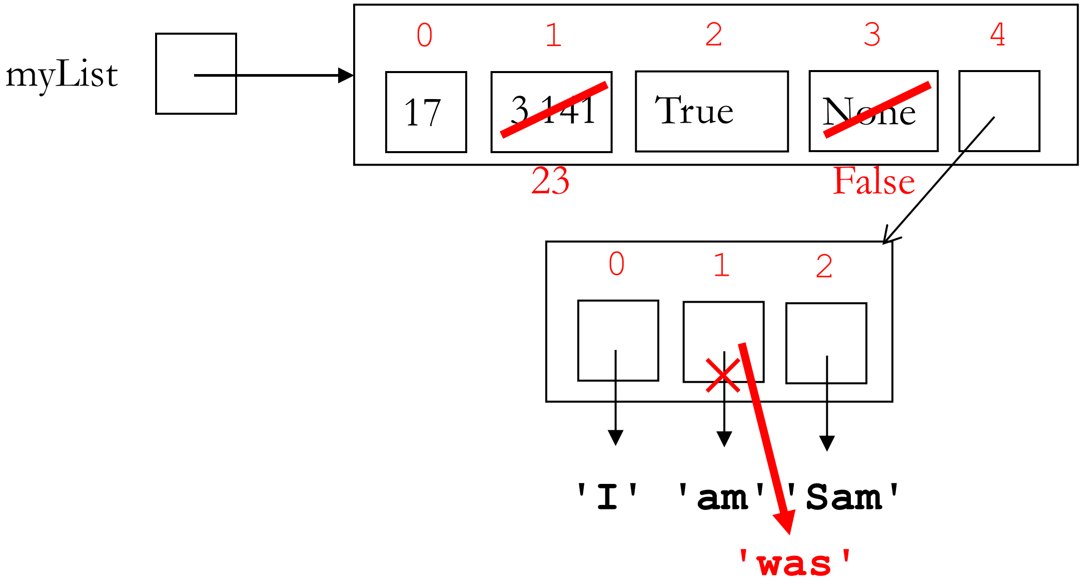
```
myList = [17, 3.141, True, None, ['I', 'am', 'Sam']]
```



List slot mutability larger example [2]

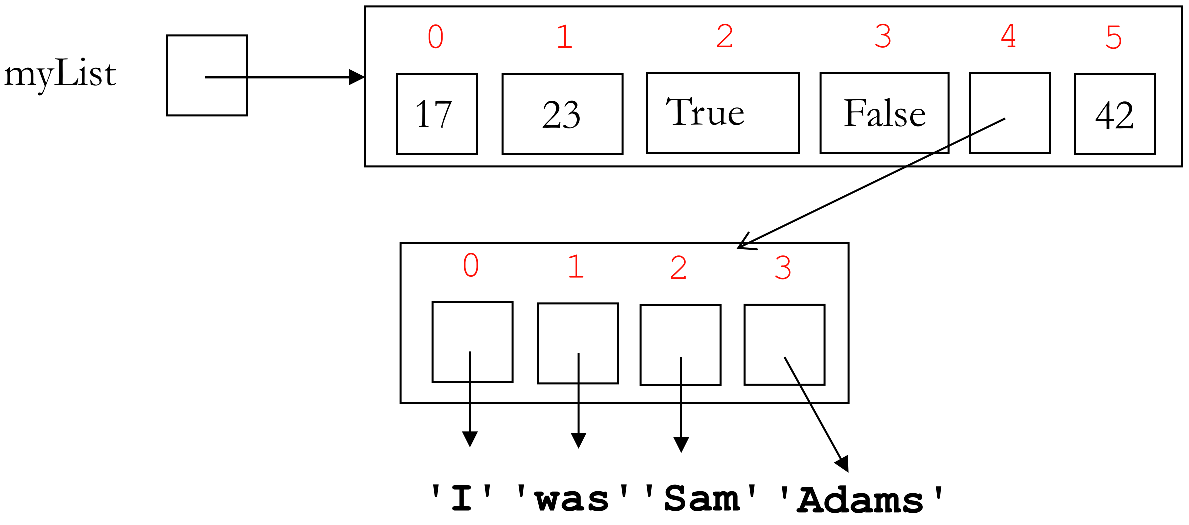
The value in any named or numbered box can change over time.
For example, the values in list slots can be changed by assignment.

```
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 Summary

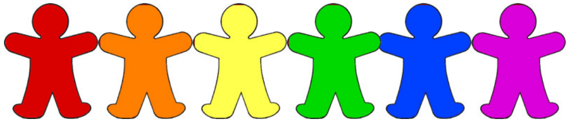
Assigning to a list index:

```
In [ ]: numStrings = ['zero', 'one', 'two', 'three', 'four']
In [ ]: numStrings[3] = 'THREE'
In [ ]: numStrings
Out[ ]: ['zero', 'one', 'two', 'THREE', 'four']
```

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

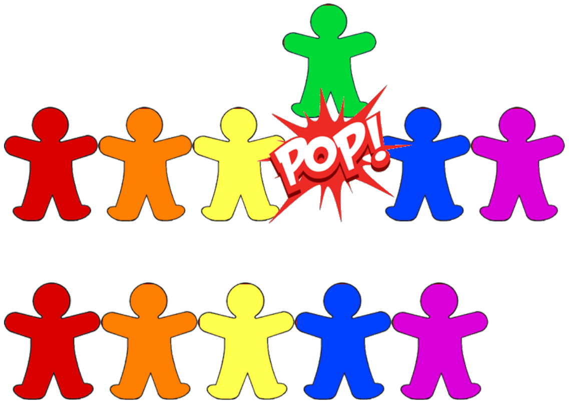
```
In [ ]: numStrings.append('five')
In [ ]: numStrings
Out[ ]: ['zero', 'one', 'two', 'THREE', 'four', 'five']
```

More list mutability



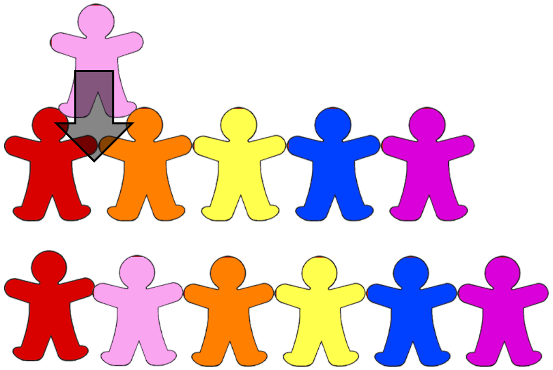
`pop`

(remove an element from a list)



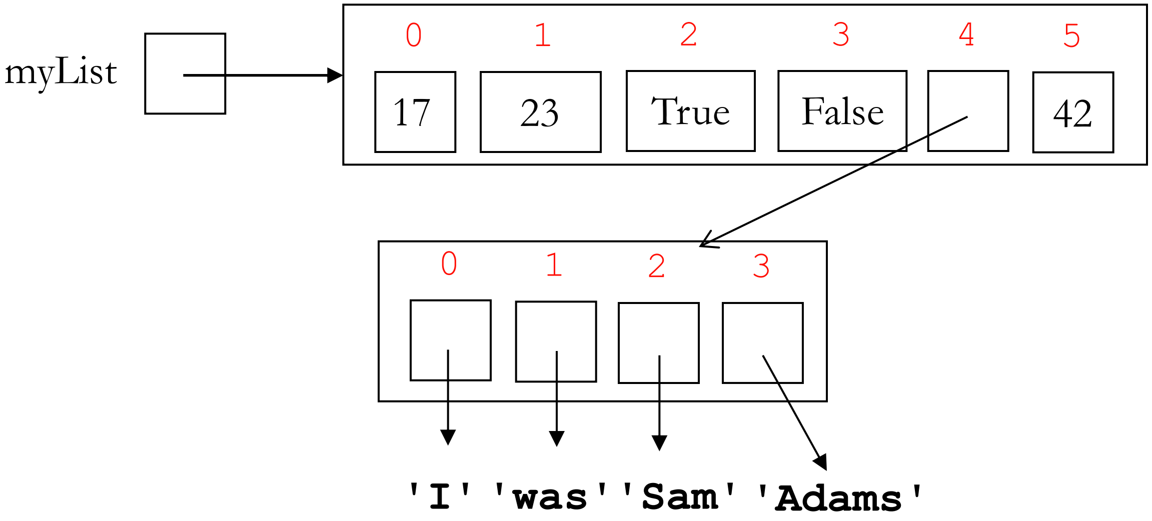
`insert`

(adding a new element to a list)



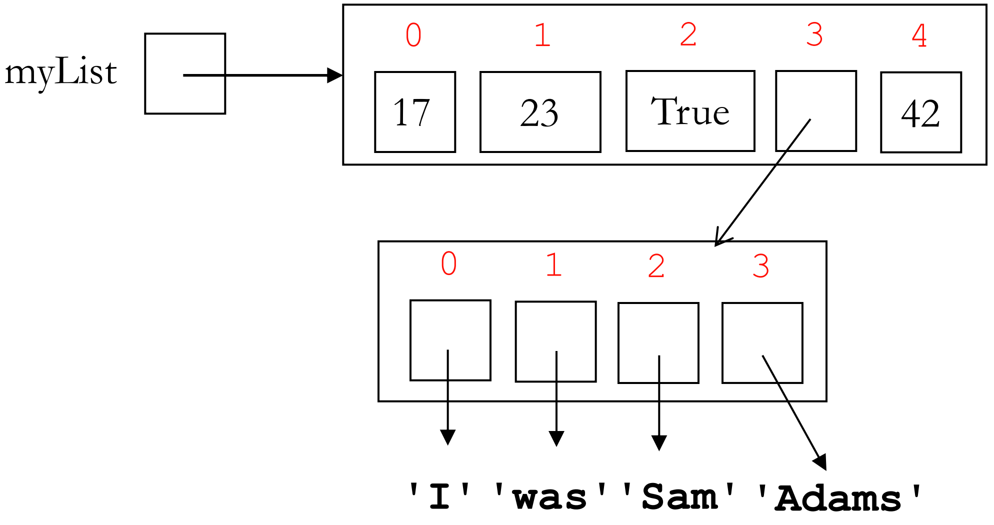
pop: remove slot at an index and return its value

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



pop: remove slot at an index and return its value

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

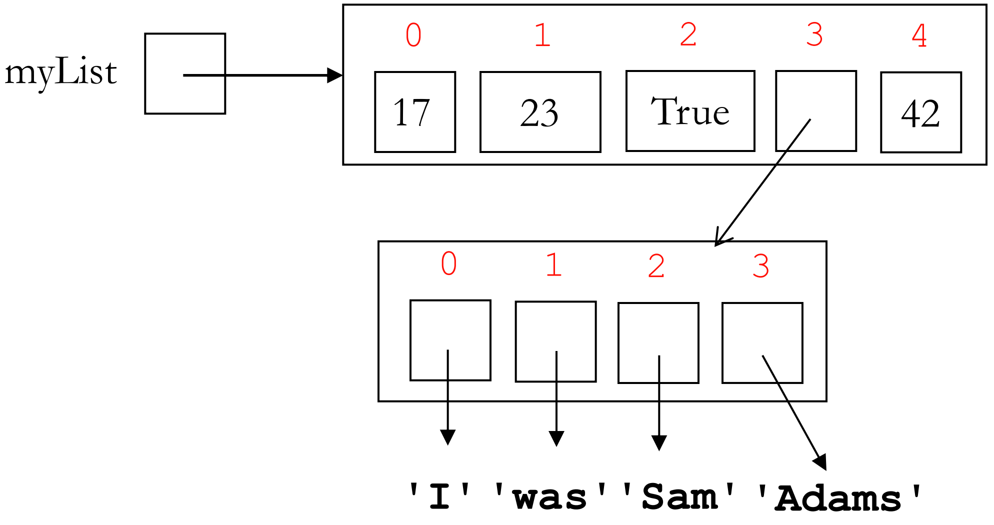


pop: remove slot at an index and return its value

```
myList.pop(3) → False
```

Indices of slots after 3 are decremented

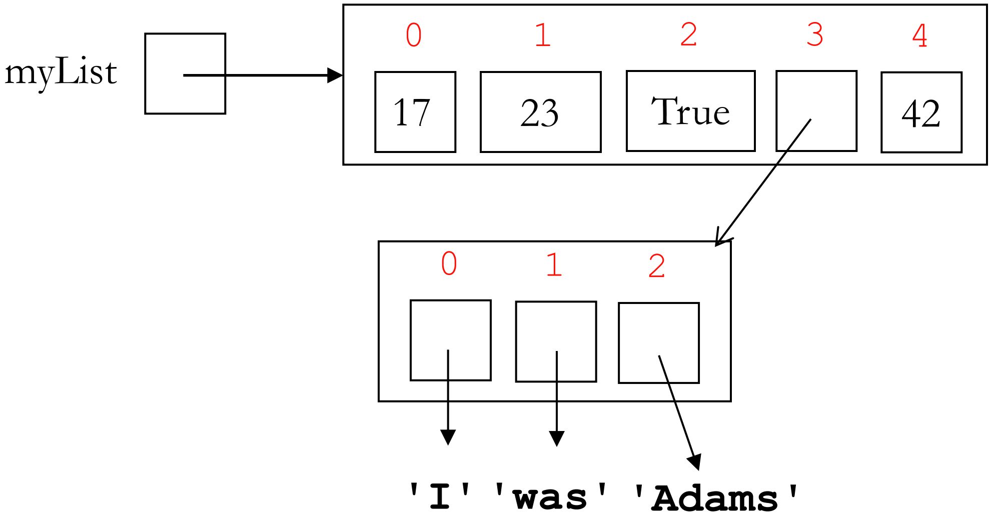
```
myList[3].pop(2)
```



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

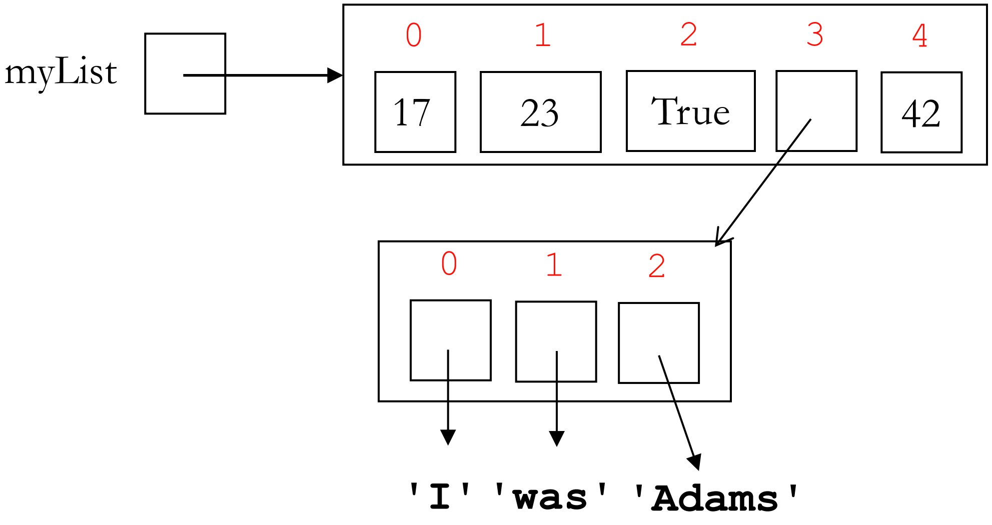


pop: remove slot at an index and return its value

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

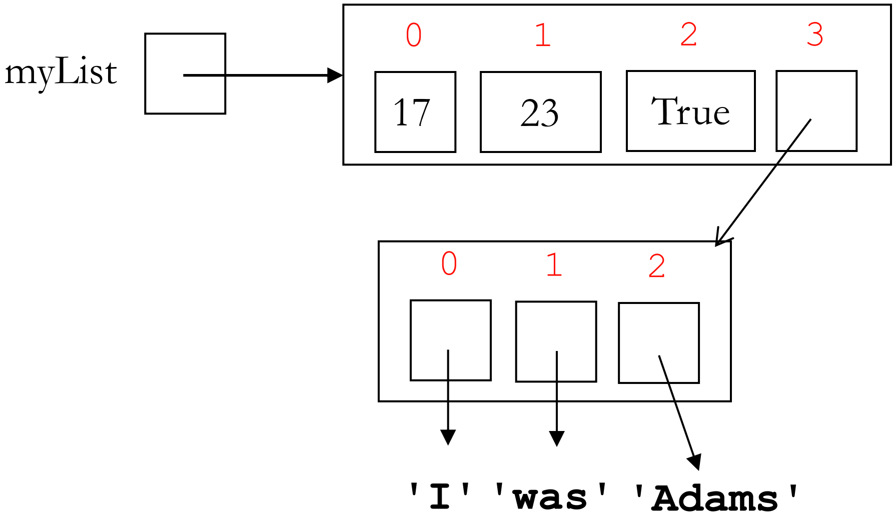
`myList[3].pop(2)` → 'Sam' # Index of previous slot 3 is decremented

`myList.pop()`



pop: remove slot at an index and return its value

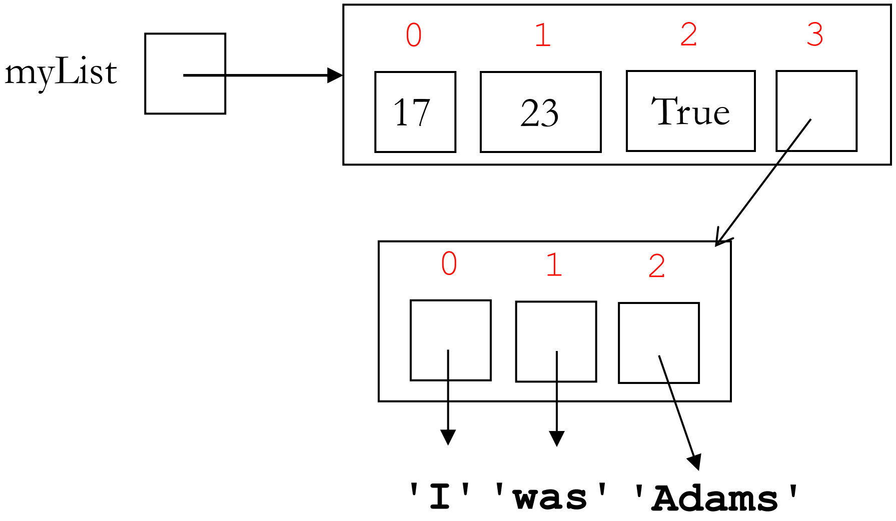
```
myList.pop(3) → False          # Indices of slots after 3 are decremented
myList[3].pop(2) → 'Sam'       # Index of previous slot 3 is decremented
myList.pop() → 42              # When no index, last one is assumed
```



insert: add a slot, add an index

Fill in the BLANK

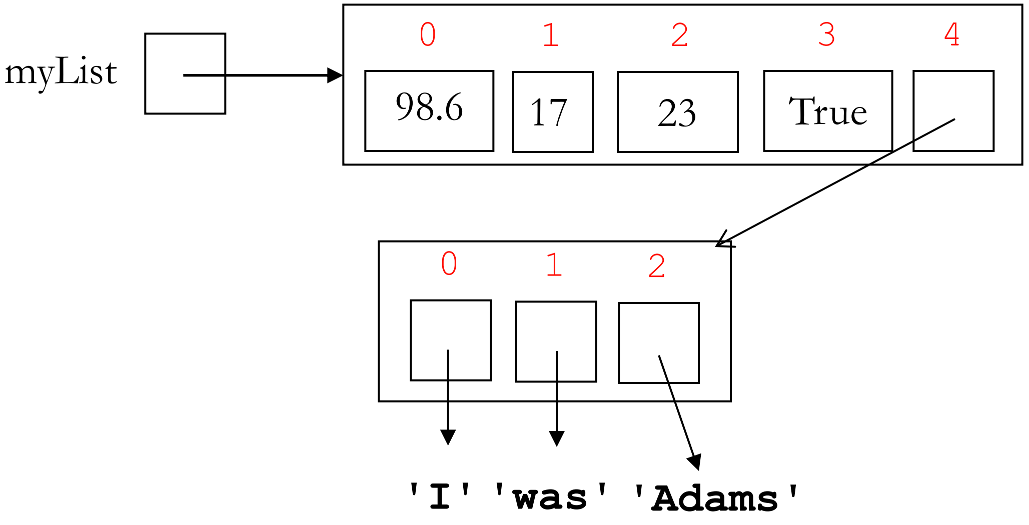
```
myList.insert(0, 98.6)
```



insert: add a slot, add an index

Fill in the BLANK

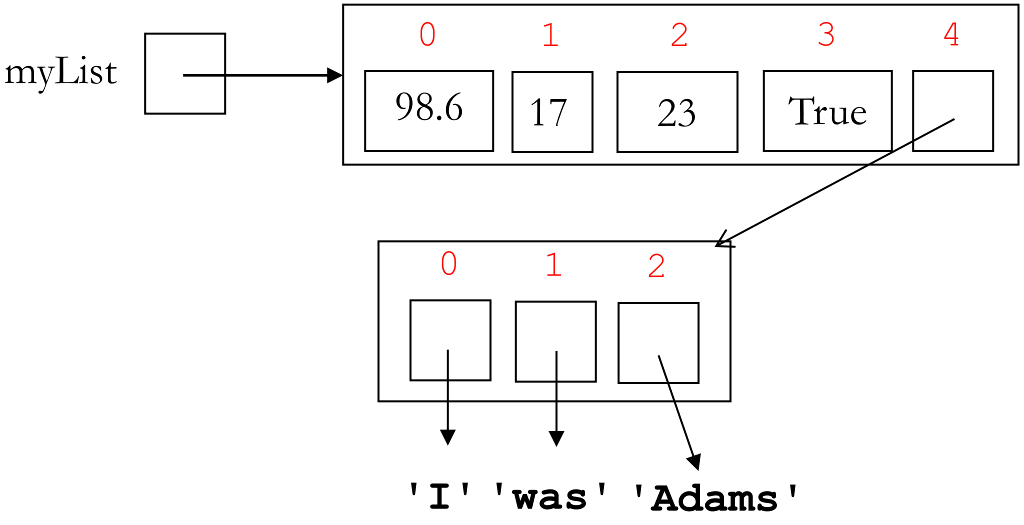
```
myList.insert(0, 98.6)
```
Indices of previous slots 0 and above
# are incremented



insert: add a slot, add an index

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

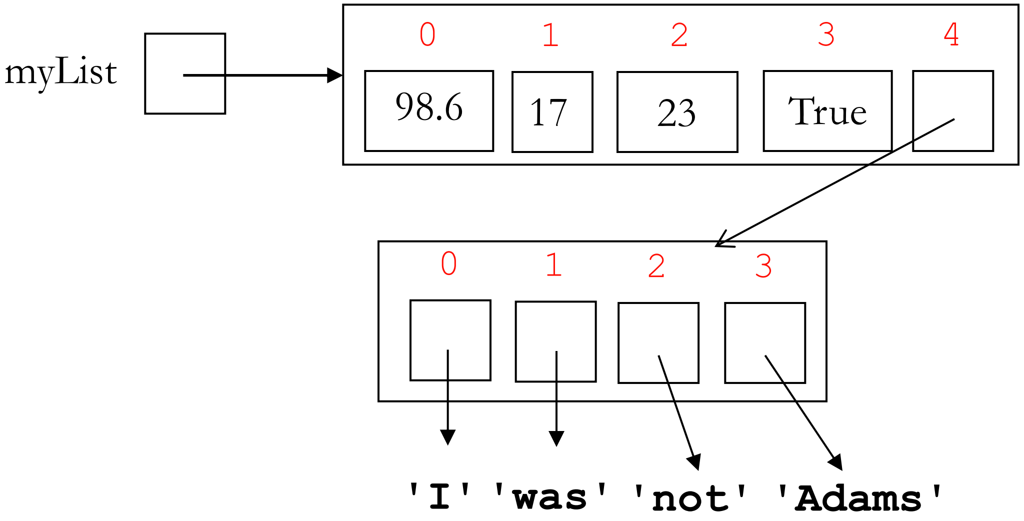
```
myList[4].insert(2, 'not')
```



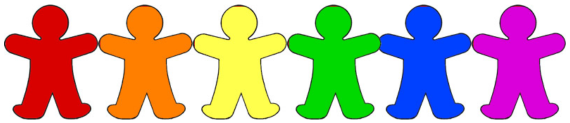
`insert`: add a slot, add an index

```
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

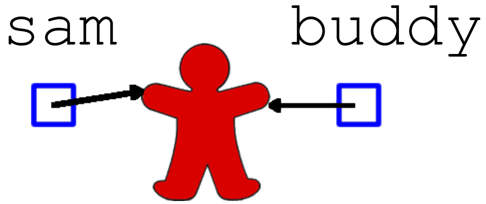


More list mutability



"Aliasing"

(same object stored in multiple variables and slots)



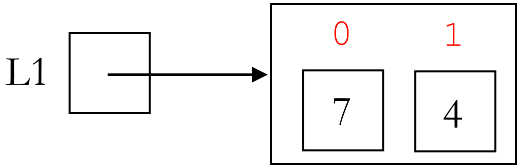
sam buddy

Aliasing: the same object can be stored in different variables & slots [1]

```
L1 = [7, 4]
```

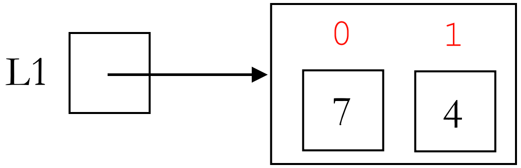
Aliasing: the same object can be stored in different variables & slots [1]

```
L1 = [7, 4]
```



Aliasing: the same object can be stored in different variables & slots [1]

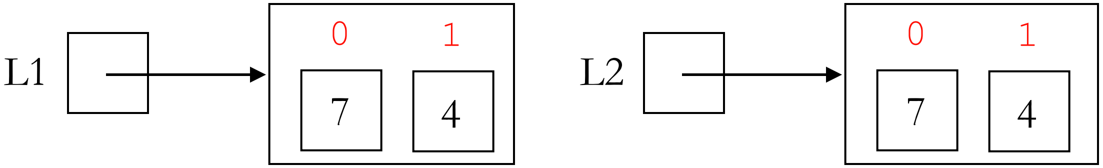
```
L1 = [7, 4]
L2 = [7, 4]
```



Aliasing: the same object can be stored in different variables & slots [1]

```
L1 = [7, 4]
```

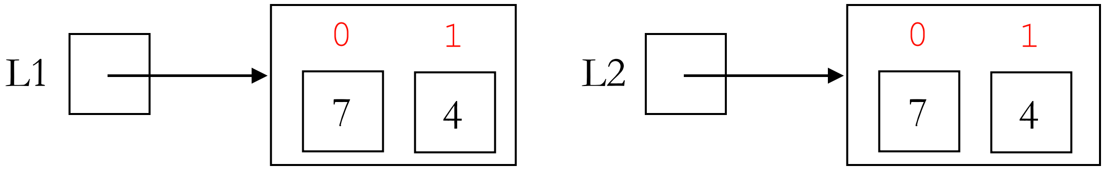
`L2 = [7, 4]` # L2 is a **copy** of L1; can also write as `L2 = L1[0:2]` or `L2 = L1[:]`



Aliasing: the same object can be stored in different variables & slots [1]

Fill in the BLANK

```
L1 = [7, 4]
L2 = [7, 4] # L2 is a copy of L1; can also write as L2 = L1[0:2] or L2 = L1[:]
L3 = L2
```

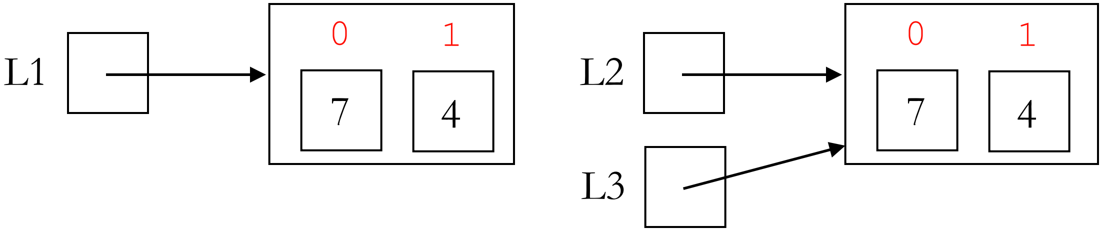


Aliasing: the same object can be stored in different variables & slots [1]

L1 = [7, 4]

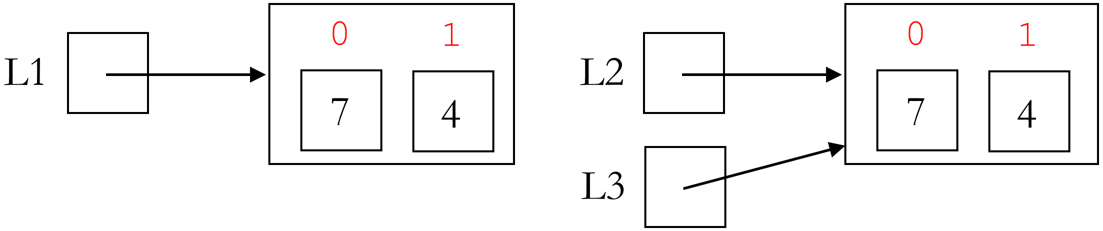
L2 = [7, 4] # L2 is a **copy** of L1; can also write as **L2 = L1[0:2]** or **L2 = L1[:]**

L3 = L2 # L3 is the **same** list object as L2



Aliasing: the same object can be stored in different variables & slots [1]

```
L1 = [7, 4]
L2 = [7, 4] # L2 is a copy of L1; can also write as L2 = L1[0:2] or L2 = L1[:]
L3 = L2   # L3 is the same list object as L2
L2[1] = 8
```



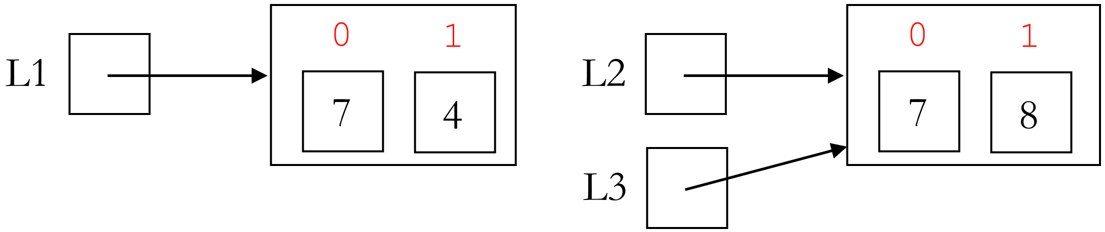
Aliasing: the same object can be stored in different variables & slots [1]

`L1 = [7, 4]`

`L2 = [7, 4]` # L2 is a **copy** of L1; can also write as **`L2 = L1[0:2]`** or **`L2 = L1[:]`**

`L3 = L2` # L3 is the **same** list object as L2

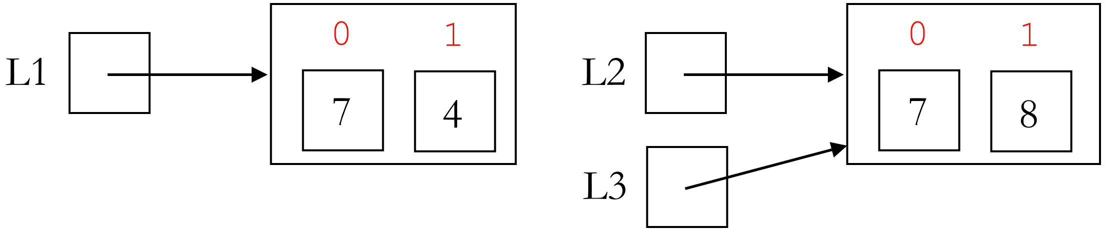
`L2[1] = 8` # Changes L2 and L3 but not L1



Aliasing: the same object can be stored in different variables & slots [1]

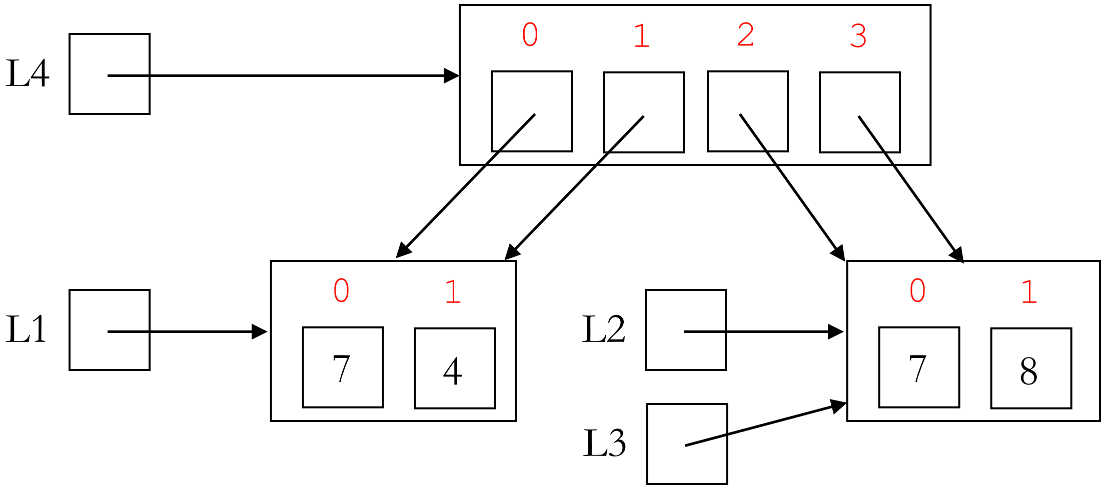
CS 111 Fill in the BLANK

```
L1 = [7, 4]
L2 = [7, 4] # L2 is a copy of L1; can also write as L2 = L1[0:2] or L2 = L1[:]
L3 = L2   # L3 is the same list object as L2
L2[1] = 8  # Changes L2 and L3 but not L1
L4 = [L1, L1, L2, L3]
```



Aliasing: the same object can be stored in different variables & slots [1]

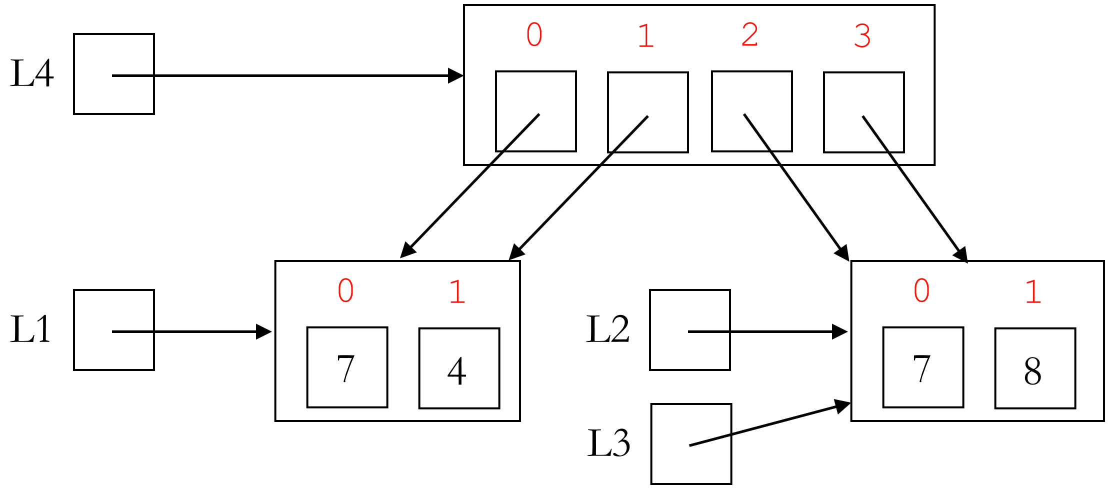
```
L1 = [7, 4]
L2 = [7, 4] # L2 is a copy of L1; can also write as L2 = L1[0:2] or L2 = L1[:]
L3 = L2  # L3 is the same list object as L2
L2[1] = 8  # Changes L2 and L3 but not L1
L4 = [L1, L1, L2, L3]  # Introduces new aliases through L4
```



Aliasing: the same object can be stored in different variables & slots [1]

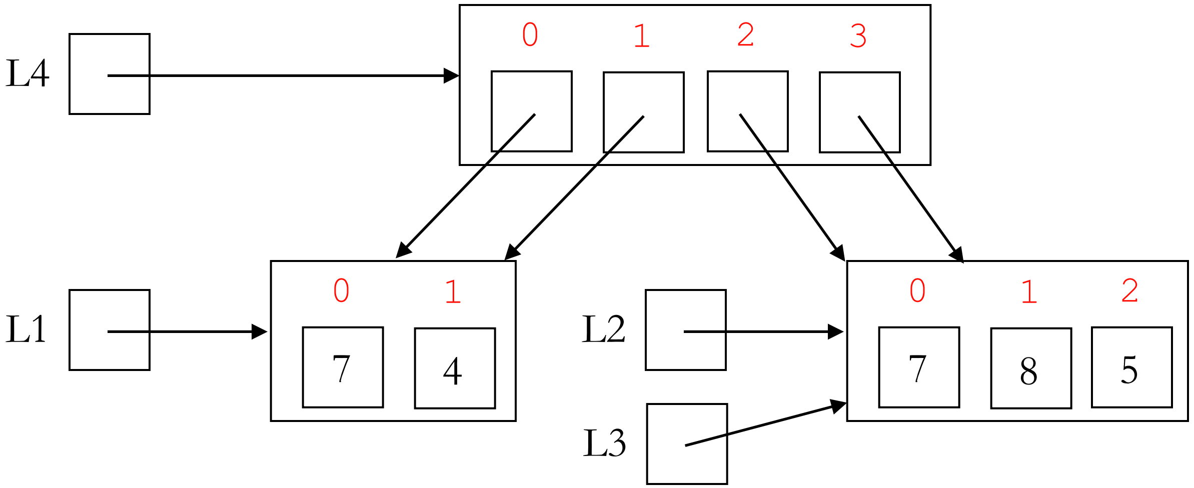
Fill in the BLANK

```
L1 = [7, 4]
L2 = [7, 4] # L2 is a copy of L1; can also write as L2 = L1[0:2] or L2 = L1[:]
L3 = L2  # L3 is the same list object as L2
L2[1] = 8  # Changes L2 and L3 but not L1
L4 = [L1, L1, L2, L3]  # Introduces new aliases through L4
L4[2].append(5)
```



Aliasing: the same object can be stored in different variables & slots [1]

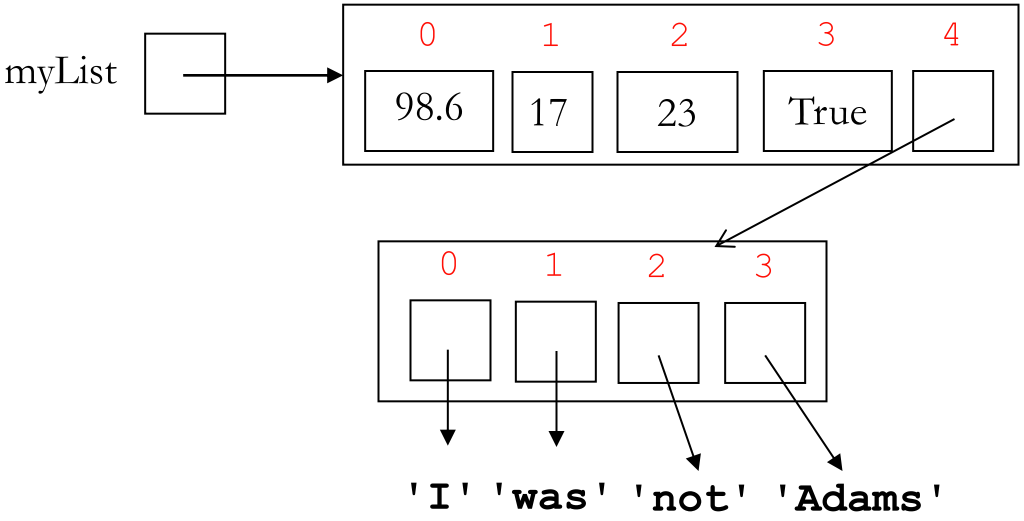
```
L1 = [7, 4]
L2 = [7, 4] # L2 is a copy of L1; can also write as L2 = L1[0:2] or L2 = L1[:]
L3 = L2  # L3 is the same list object as L2
L2[1] = 8  # Changes L2 and L3 but not L1
L4 = [L1, L1, L2, L3]  # Introduces new aliases through L4
L4[2].append(5) # Changes L2, L3, L4[2], and L4[3], but not L1, L4[0], and L4[1]
```



Aliasing: the same object can be stored in different variables & slots [2]

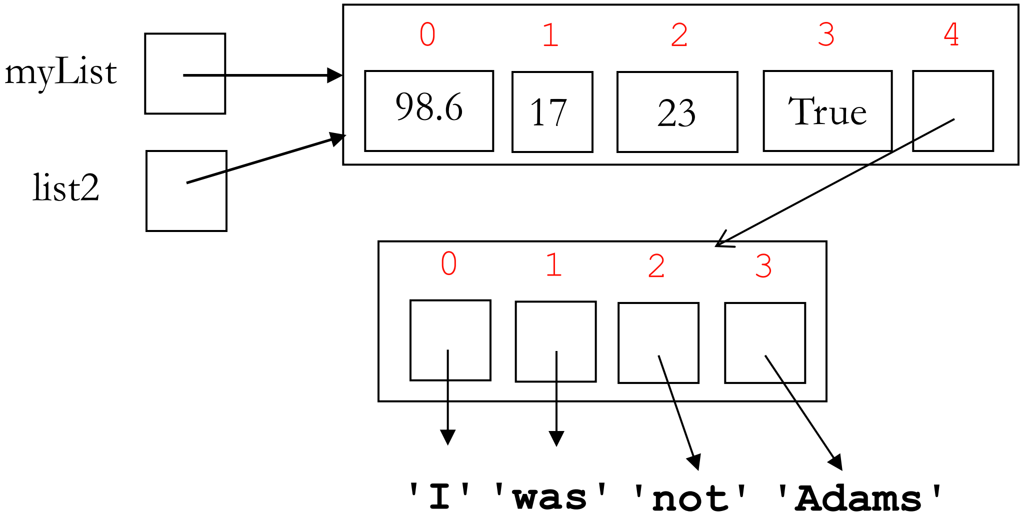
Fill in the BLANK

```
list2 = myList
```



Aliasing: the same object can be stored in different variables & slots [2]

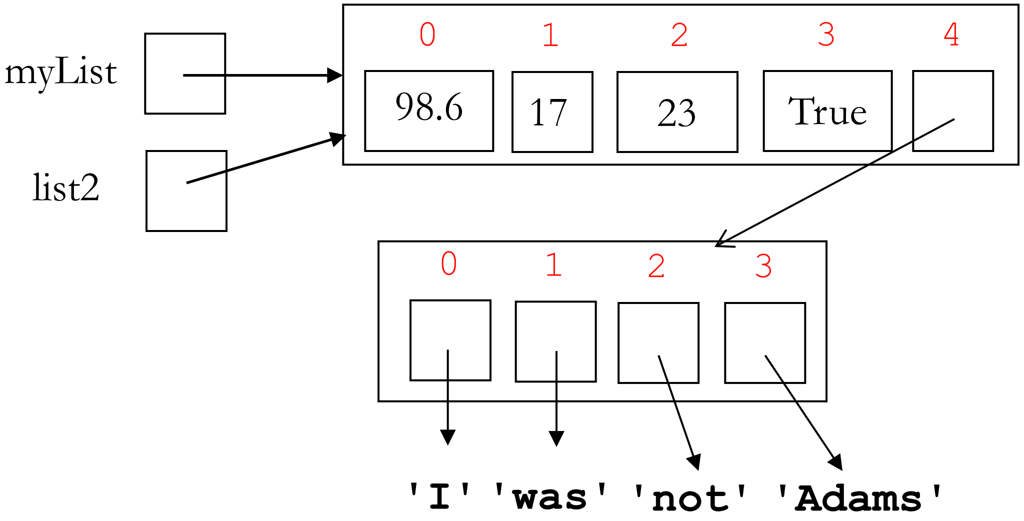
```
list2 = myList
```
list2 and myList are now the **same** list, not just copies



Aliasing: the same object can be stored in different variables & slots [2]

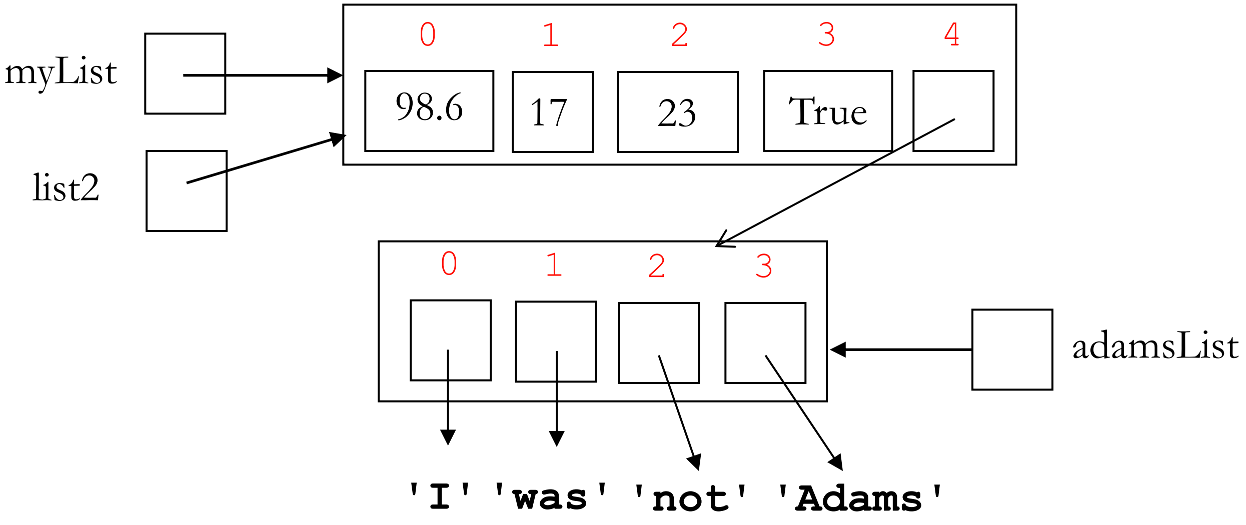
Fill in the BLANK

```
list2 = myList    # list2 and myList are now the same list, not just copies
adamsList = list2[4]
```



Aliasing: the same object can be stored in different variables & slots [2]

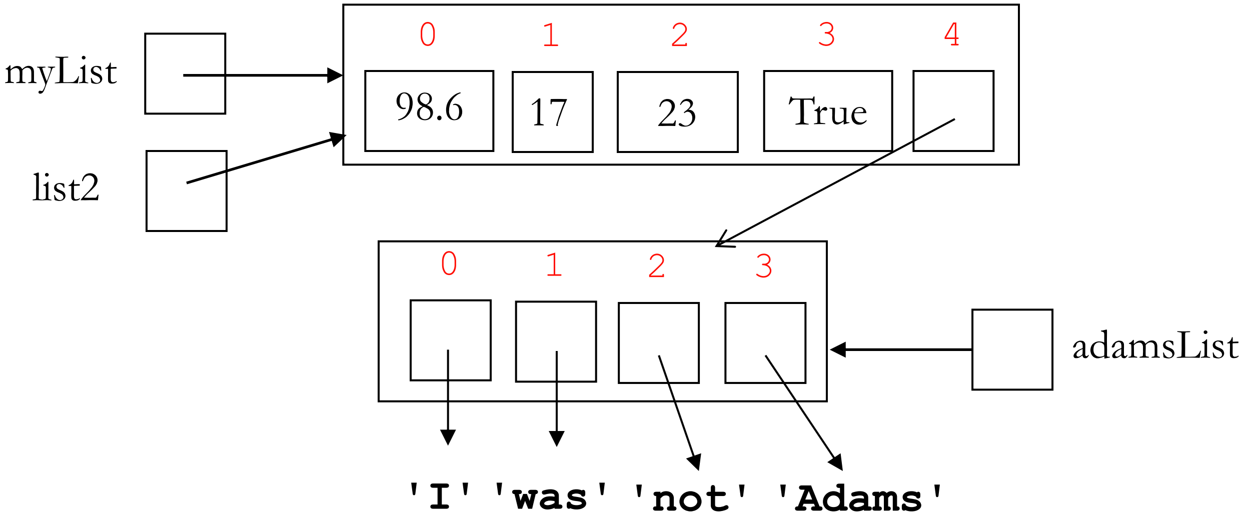
```
list2 = myList   # list2 and myList are now the same list, not just copiesx
adamsList = list2[4]  # Now myList[4] is also an alias for adamsList
```



Aliasing: the same object can be stored in different variables & slots [2]

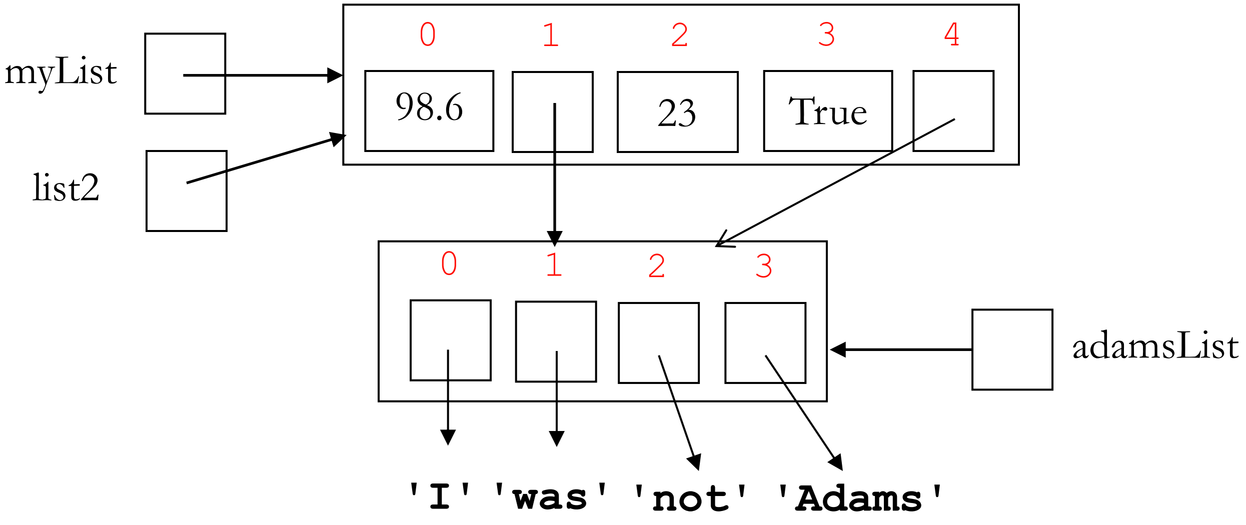
Fill in the BLANK

```
list2 = myList      # list2 and myList are now the same list, not just copies
adamsList = list2[4]  # Now myList[4] is also an alias for adamsList
myList[1] = myList[4]
```



Aliasing: the same object can be stored in different variables & slots [2]

```
list2 = myList          # list2 and myList are now the same list, not just copies
adamsList = list2[4]    # Now myList[4] is also an alias for adamsList
myList[1] = myList[4]   # Now list2[1] is another alias for adamsList
```



Aliasing: the same object can be stored in different variables & slots [2]

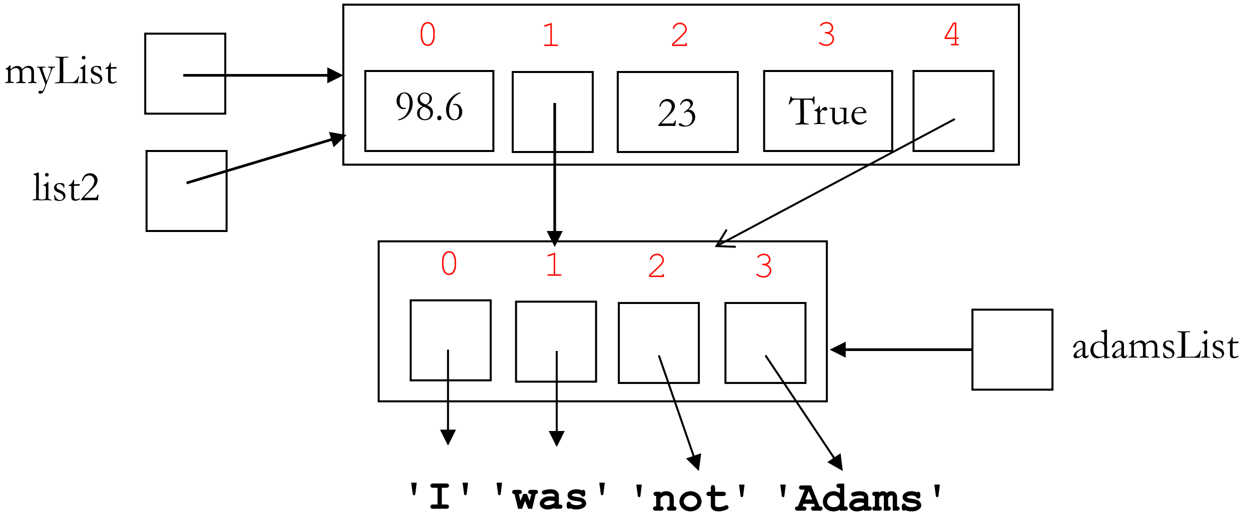
Fill in the BLANK

```
list2 = myList
```
# list2 and myList are now the **same** list, not just copies

```
adamsList = list2[4]
```
# Now myList[4] is **also** an alias for adamsList

```
myList[1] = myList[4]
```
# Now list2[1] is another alias for adamsList

```
adamsList[2] = 'JQ'
```



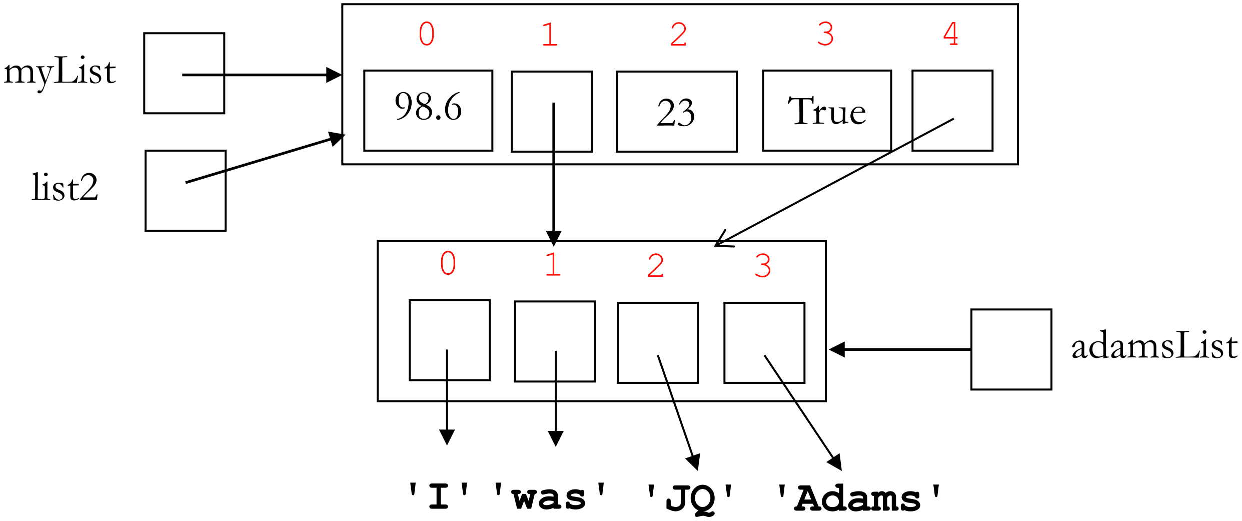
Aliasing: the same object can be stored in different variables & slots [2]

```
list2 = myList
```
# list2 and myList are now the **same** list, not just copies

```
adamsList = list2[4]
```
# Now myList[4] is **also** an alias for adamsList

```
myList[1] = myList[4]
```
# Now list2[1] is another alias for adamsList

```
adamsList[2] = 'JQ'
```
Because of aliasing, this also changes myList[1][2],
# myList[4][2], list2[1][2], and list2[4][2]

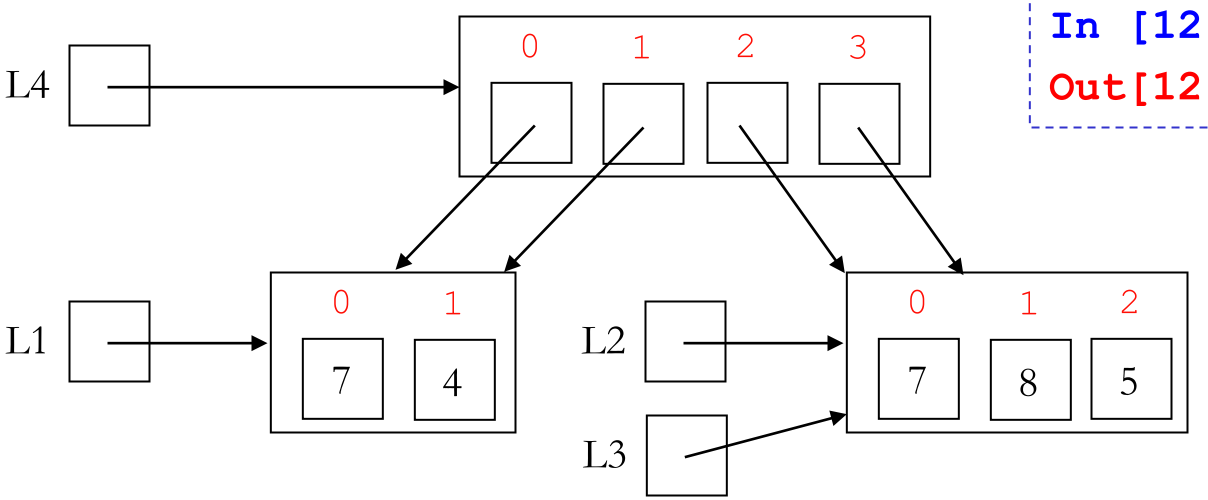


Built-in `id` function identifies which lists are the same

The built-in **`id`** function returns a unique number for every object in memory. You can think of it as an abstract address for that object. You can use it to tell which objects are the "same" objects in memory.

```
In [6]: id(L1)
Out[6]: 140689397337616

In [7]: id(L2)
Out[7]: 140689376966528

In [8]: id(L3)
Out[8]: 140689376966528
```

```
In [9]: id(L4[0])
Out[9]: 140689397337616

In [10]: id(L4[1])
Out[10]: 140689397337616

In [11]: id(L4[2])
Out[11]: 140689376966528

In [12]: id(L4[3])
Out[12]: 140689376966528
```

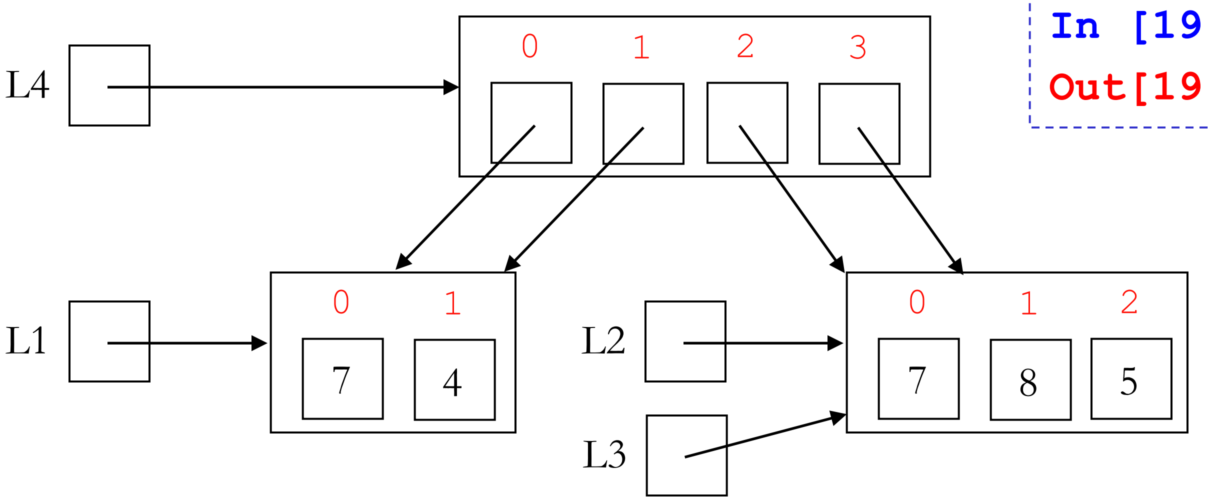


Built-in `is` operator indicates which lists are the same

The built-in binary **`is`** operator returns **`True`** if its operands have the same **`id`** and **`False`** otherwise. It's an easy way to test whether two list objects are the same.

```
In [13]: L1 is L2
Out[13]: False

In [14]: L1 is L4[0]
Out[14]: True

In [15]: L2 is L3
Out[15]: True
```

```
In [16]: L2 is L4[0]
Out[16]: False

In [17]: L3 is L4[2]
Out[17]: True

In [18]: L4[0] is L4[1]
Out[18]: True

In [19]: L4[1] is L4[2]
Out[19]: False
```



What is the final value of `c[0]`?

it's your turn

```
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!

it's your turn

```
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]
```

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

it's your turn

```
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]
```

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

Lists are mutable. What about strings?

Strings are sequences:

```
In [20]: name = 'Gryffindor'
In [21]: name[2]         # 'y'
In [22]: name[4:8]       # 'find'
In [23]: 'do' in name    # True
```

Mutation operations **do not work** on strings:

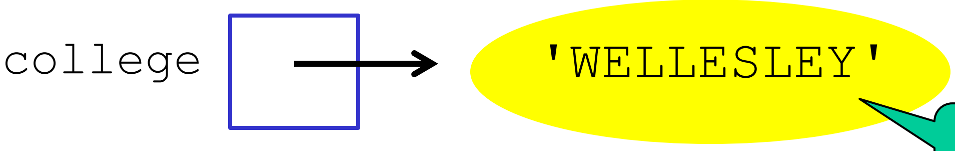
```
In [24]: name[4] = 't' # what happens?
---------------------------------------------------------------------------
TypeError … name[0]= ’t’
TypeError: 'str' object does not support item assignment
```

```
In [25]: name.append('s') # what happens?
---------------------------------------------------------------------------
AttributeError … name.append(’s’)
AttributeError: 'str' object has no attribute 'append'
```

Strings are immutable sequences

Once you create a string, it cannot be changed

```
In[26]: college = 'WELLESLEY'
```

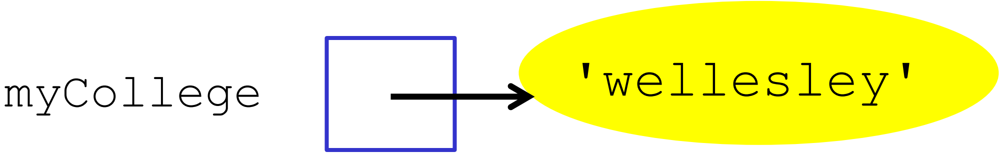


college

'WELLESLEY'

Immutable, not changed.

```
In[27]: college.lower( )
Out[27]: 'wellesley'  # Returns a new string 'wellesley';
                      #  old one is unchanged!

In[28]: myCollege = college.lower( )
```



myCollege

'wellesley'

Tuples

Lists are **mutable sequences** of values.
Tuples are **immutable sequences** of values.

Tuples are written as comma-separated values delimited by parentheses.

```
# A homogeneous tuple of five integers    (a 4-tuple)
(5, 8, 7, 1, 3)

# A homogeneous tuple of four strings
('Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin')

# A heterogeneous tuple of three elements  (a 3-tuple)
(42, 'Hello', False)

# A pair is a tuple with two elements      (a 2-tuple)
(7, 3)

(7, )  # A tuple with one element must use a comma to avoid
       # being confused with a parenthesized expression

()    # A tuple with 0 values
```

Tuple Assignment

Suppose `harryInfo` is a tuple of three values:

```
In [46]: harryInfo = ('Harry Potter', 11, True)
```

Then we can extract three named values from `harryInfo` by a single assignment to a tuple of three variable names:

```
In [47]: (name, age, glasses) = harryInfo
```

This so-called **tuple assignment** is just a shorthand for three separate assignments:

```
name = harryInfo[0]
age = harryInfo[1]
glasses = harryInfo[2]
```

We can now use these names like any other variables:

```
In [48]: print(name.lower(), age + 6, not glasses)

harry potter 17 False
```

Parens are not necessary in a tuple assignment; above, we could also have written:

```
In [49]: name, age, glasses = harryInfo
```

Tuples are also immutable sequences

Like strings, tuples support all sequence operations that do not involve mutation.

```
In[32]: houseTuple = ('Gryffindor', 'Hufflepuff',
                      'Ravenclaw', 'Slytherin')
In[33]: houseTuple[0]
Out[33]:'Gryffindor'

In[34]: houseTuple[1:3]
Out[34]: ('Hufflepuff', 'Ravenclaw')

In[35]: houseTuple.count('Slytherin')
Out[35]: 1

In[36]: 'Ravenclaw' in houseTuple
Out[36]: True

In[37]: houseTuple * 2 + ('12 Grimmauld Place',)
Out[37]:('Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin',
          'Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin',
          '12 Grimmauld Place')
```

Mutation operations do not work on tuples

```
In [38]: houseTuple[0] = '4 Privet Drive'
----------------------------------------------------------
TypeError … houseTuple[0] = '4 Privet Drive'
TypeError: 'tuple' object does not support item assignment

In [39]: houseTuple.append('The Shrieking Shack')
-----------------------------------------------------------
AttributeError … houseTuple.append('The Shrieking Shack')
AttributeError: 'tuple' object has no attribute 'append'

In [40]: houseTuple.pop(1)
-----------------------------------------------------------
AttributeError … houseTuple.pop(1)
AttributeError: 'tuple' object has no attribute 'pop'
```

Conversion between sequence types

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

```
In [41]: word = "Wellesley"
In [42]: list(word)
Out[42]: ['W', 'e', 'l', 'l', 'e', 's', 'l', 'e', 'y']

In [43]: tuple(word)
Out[43]: ('W', 'e', 'l', 'l', 'e', 's', 'l', 'e', 'y')

In [44]: numbers = range(5, 15, 2)
In [45]: str(numbers)
Out[45]: 'range(5, 15, 2)'
```

Enumerations

Digging Deeper

When called on a sequence, the **enumerate** function returns a sequence of **pairs** of indices and values.

```
In [50]: list(enumerate('boston'))
Out[50]: [(0, 'b'), (1, 'o'), (2, 's'), (3, 't'), (4, 'o'), (5, 'n')]

In [51]: list(enumerate([7, 2, 8, 5]))
Out[51]: [(0, 7), (1, 2), (2, 8), (3, 5)]


In [52]: for (index, char) in enumerate('boston'):
...         print(index, char)
0 b
1 o
2 s
3 t
4 o
5 n
```

Note that `for (index, char) in` is a use of tuple assigment notation in a **for** loop.

Test your knowledge

1. What are the different ways to create lists? What can be passed into the `list()` function?
2. Define mutable and identify whether strings, lists, and ranges are mutable.
3. Explain how the methods `pop()`, `insert()`, and `append()` change lists when the arguments to those methods are numbers, strings, or elements of the same list or other list.
4. Does ordering matter in lists? Explain why or why not. What is the result of `[1, 2, 3] == [3, 2, 1]`?
5. Why don't the methods `pop()`, `insert()`, and `append()` work on strings?
6. What does the `id()` function do? How can it be used to determine aliasing?
7. On slide 46, how would you memory diagram and result change if `b = a` instead of `b = [15, 20]`?
8. What are the similarities and differences between tuples and lists? Why might you use one over the other?
9. The above slides did not discuss iteration over tuples. Do you think this is possible? Why or why not?
10. What are the advantages of enumeration? In what context, would it be useful to use enumeration?