LISTS, MUTABILITY

(download slides and .py files to follow along)

6.100L Lecture 10

Ana Bell

INDICES and ORDERING in LISTS

```
a_list = []
L = [2, 'a', 4, [1,2]]
```

empty list

`len(L)` → evaluates to 4

`L[0]` → evaluates to 2

Indexing starts at 0

`L[3]` → evaluates to `[1,2]`, another list!

`[2,'a'] + [5,6]` → evaluates to `[2,'a',5,6]`

`max([3,5,0])` → evaluates to 5

`L[1:3]` → evaluates to `['a', 4]`

Slicing just like strings

`for e in L` → loop variable becomes each element in L

`L[3] = 10` → mutates `L` to `[2,'a',4,10]`

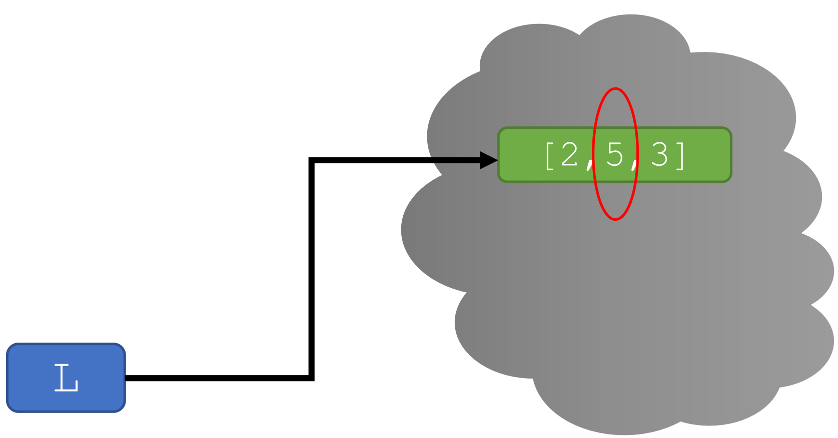
Mutate L by changing an element

MUTABILITY

- Lists are **mutable**!
- Assigning to an element at an index **changes** the value

```
L = [2, 4, 3]
L[1] = 5
```

- `L` is now `[2, 5, 3]`; note this is the **same object** `L`

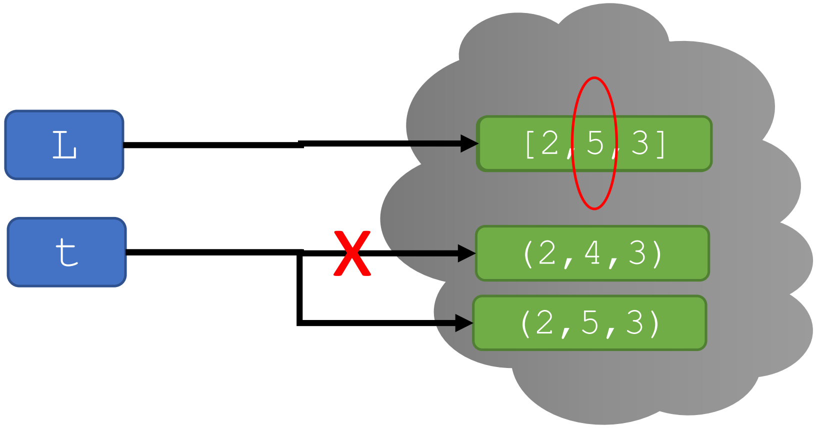


different from strings and tuples!

MUTABILITY

- Compare
  - Making L by **mutating an element** vs.
  - Making t by **creating a new object**

```
L = [2, 4, 3]
L[1] = 5
t = (2, 4, 3)
t = (2, 5, 3)
```



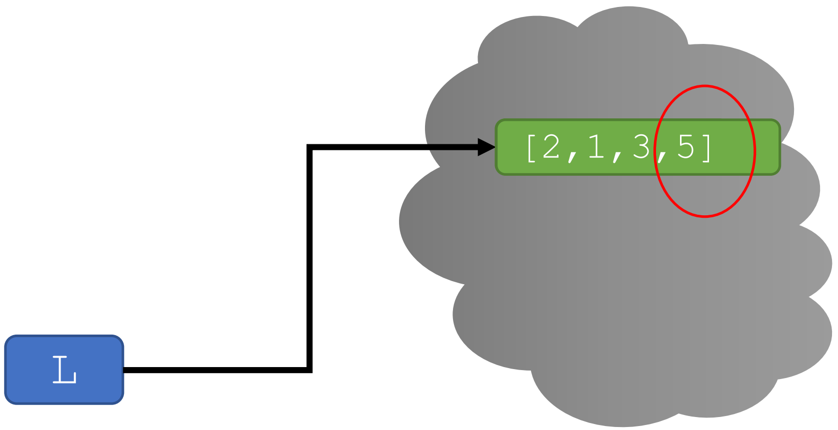
OPERATION ON LISTS – append

Function name

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

L and element are your objects

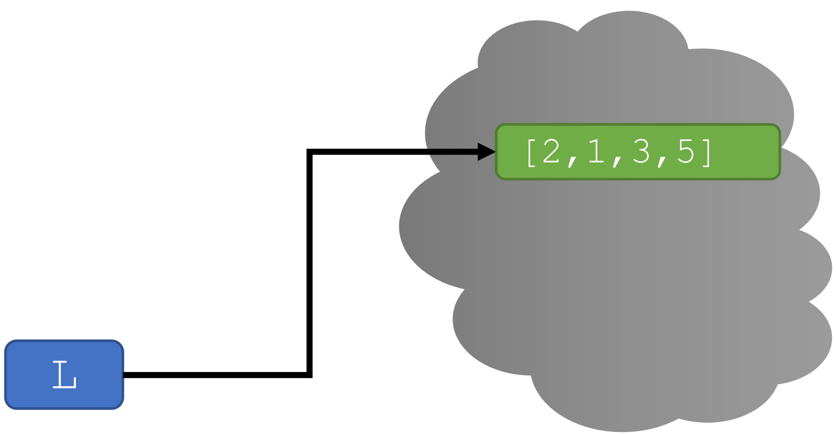
```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`



OPERATION ON LISTS – append

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`
```
L = L.append(5)
```

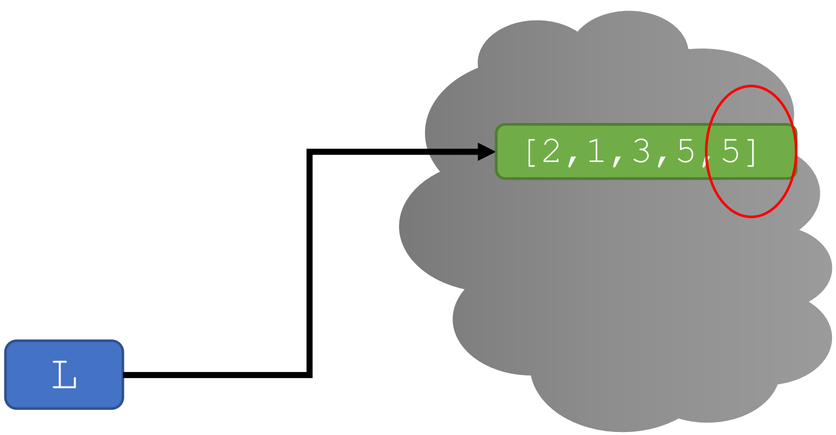


OPERATION ON LISTS – append

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`

```
L = L.append(5)
```



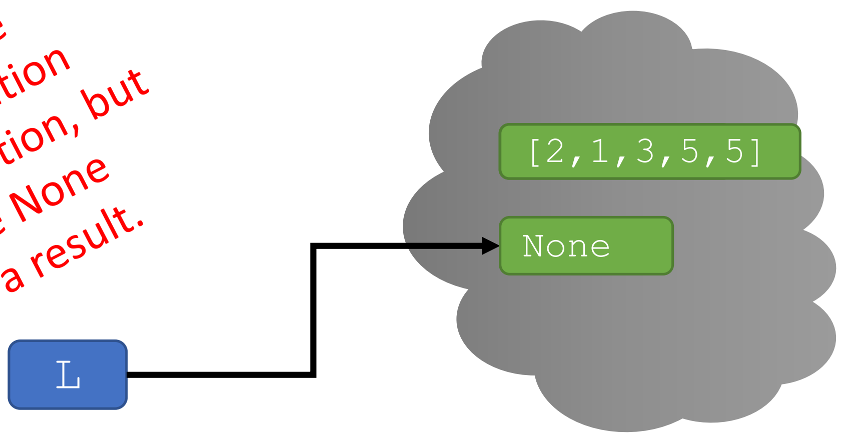
OPERATION ON LISTS – append

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`

$\boxed{\texttt{L = L.append(5)}}$

Be careful! The append operation does a mutation, but returns the None object as a result.

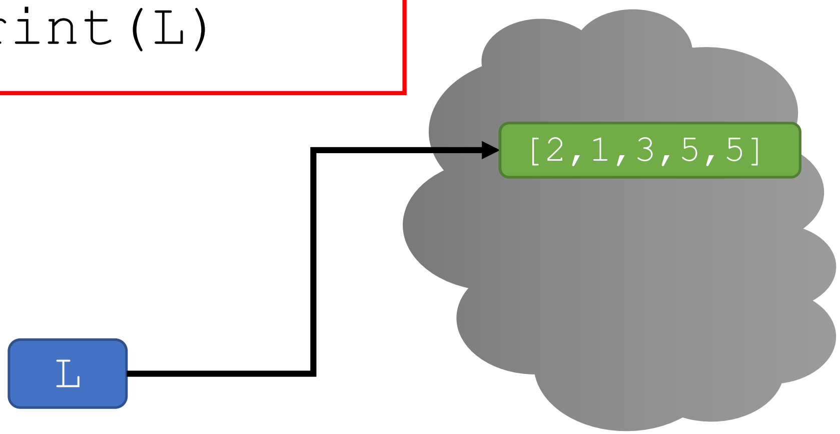


OPERATION ON LISTS – append

- **Add** an element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`

```
L.append(5)
print(L)
```
→ L is now `[2,1,3,5,5]`



Append is used strictly for its **side effect**

YOU TRY IT!

- What is the value of L1, L2, L3 and L at the end?

```
L1 = ['re']
L2 = ['mi']
L3 = ['do']
L4 = L1 + L2
L3.append(L4)
L = L1.append(L3)
```

BIG IDEA

Some functions mutate the list and don't return anything.

We use these functions for their side effect.

OPERATION ON LISTS: append

- `L = [2,1,3]`
  `L.append(5)`

function arguments

a function that works on an object of this type

an object of some type

- What is the dot?
  - Lists are Python objects, everything in Python is an object
  - Objects have **data**
  - Object types also have **associated operations**
  - Access this information by `object_name.do_something()`
  - Equivalent to calling `append` with arguments `L` and `5`

YOU TRY IT!

- Write a function that meets these specs:

```
def make_ordered_list(n):
    """ n is a positive int
    Returns a list containing all ints in order
    from 0 to n (inclusive)
    """
```

YOU TRY IT!

- Write a function that meets the specification.

```
def remove_elem(L, e):
    """
    L is a list
    Returns a new list with elements in the same order as L
    but without any elements equal to e.
    """


L = [1,2,2,2]
print(remove_elem(L, 2))   # prints [1]
```

STRINGS to LISTS

- Convert **string to list** with `list(s)`
  - Every character from `s` is an element in a list
- Use `s.split()`, to **split a string on a character** parameter, splits on spaces if called without a parameter

```
s = "I<3 cs &u?"             → s is a string
L = list(s)                  → L is ['I','<','3',' ','c','s',' ','&','u','?']

L1 = s.split(' ')            → L1 is ['I<3','cs','&u?']
L2 = s.split('<')            → L2 is ['I', '3 cs &u?']
```

LISTS to STRINGS

- Convert a **list of strings back to string**
- Use `''.join(L)` to turn a **list of strings into a bigger string**
- Can give a character in quotes to add char between every element

```
L = ['a','b','c']              → L is a list
A = ''.join(L)                 → A is "abc"
B = '_'.join(L)                → B is "a_b_c"
C = ''.join([1,2,3])           → an error
C = ''.join(['1','2','3']      → C is "123" a string!
```

YOU TRY IT!

- Write a function that meets these specs:

```
def count_words(sen):
    """ sen is a string representing a sentence
    Returns how many words are in s (i.e. a word is a
    a sequence of characters between spaces. """

print(count_words("Hello it's me"))
```

A FEW INTERESTING LIST OPERATIONS

- **Add** an element to end of list with `L.append(element)`
  - **mutates** the list
- `sort()`
  - `L = [4,2,7]`
    `L.sort()`
  - **Mutates** L
- `reverse()`
  - `L = [4,2,7]`
    `L.reverse()`
  - **Mutates** L
- `sorted()`
  - `L = [4,2,7]`
  - `L_new = sorted(L)`
  - Returns a sorted version of L (**no mutation**!)

Remember . notation: object.operation()
Do `append` operation on `L`, with
parameter `element`

MUTABILITY

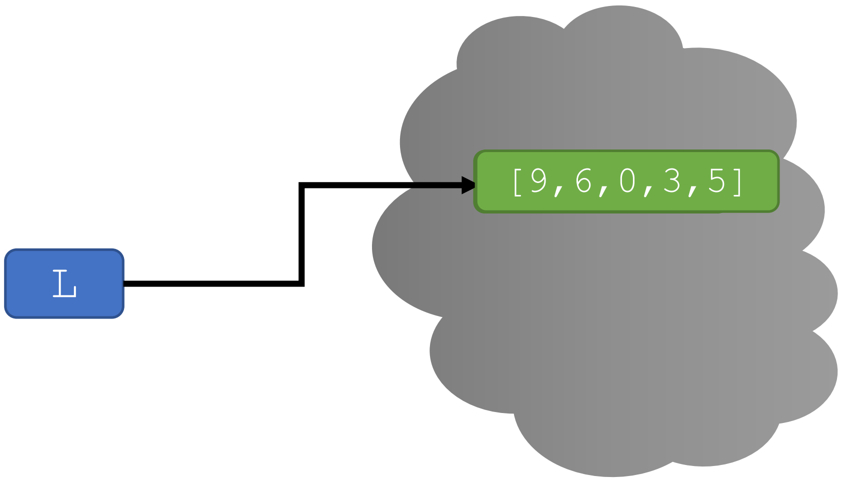
append, sort, reverse
all used for side effect

```
L=[9,6,0,3]
L.append(5)
```

`a = sorted(L)` → returns a **new** sorted list, does **not mutate** `L`

`b = L.sort()` → **mutates** `L` to be `[0,3,5,6,9]` and returns None

`L.reverse()` → **mutates** `L` to be `[9,6,5,3,0]` and returns None



MUTABILITY

`append, sort, reverse`
all used for side effect

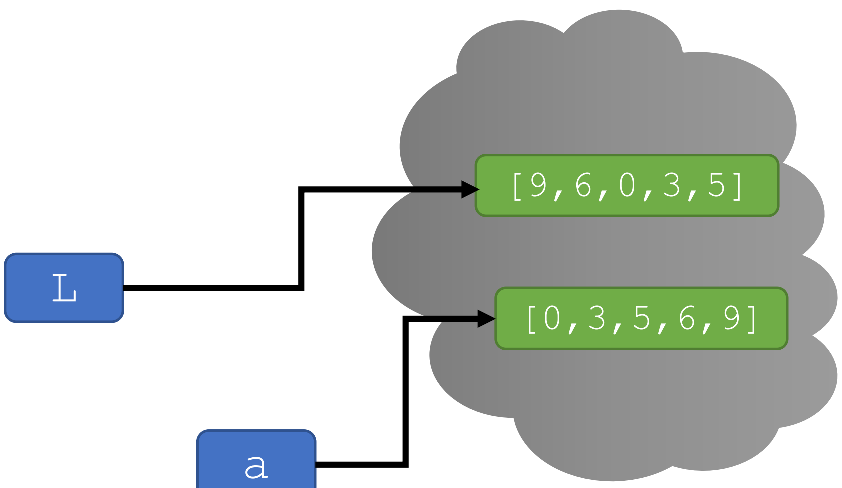
`L=[9,6,0,3]`

`L.append(5)`

`a = sorted(L)` → returns a **new** sorted list, does **not mutate** `L`

`b = L.sort()` → **mutates** `L` to be `[0,3,5,6,9]` and returns None

`L.reverse()` → **mutates** `L` to be `[9,6,5,3,0]` and returns None



MUTABILITY

`append, sort, reverse`
all used for side effect

`L=[9,6,0,3]`

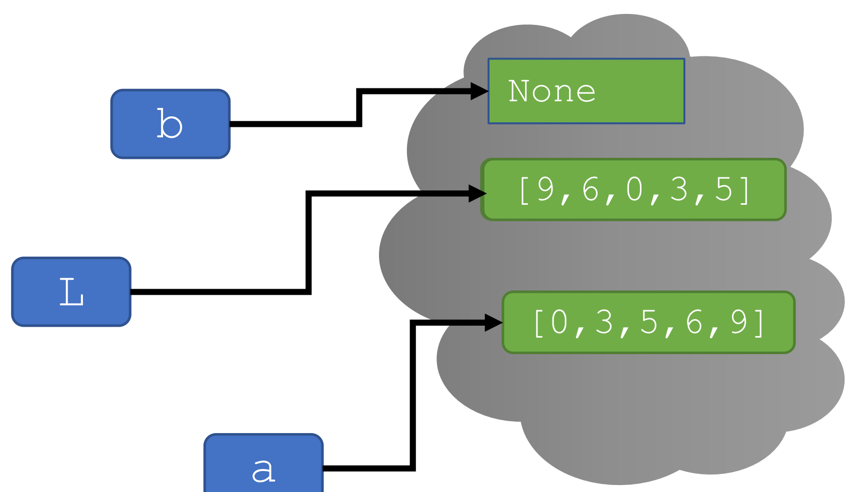
`L.append(5)`

`a = sorted(L)` → returns a **new** sorted list, does **not mutate** `L`

$\boxed{\texttt{b = }\boxed{\texttt{L.sort()}}}$ → **mutates** `L` to be `[0,3,5,6,9]` and returns None

`L.reverse()` → **mutates** `L` to be `[9,6,5,3,0]` and returns None

Never do this.
Just use L.sort()!



MUTABILITY

`append`, `sort`, `reverse`
all used for side effect

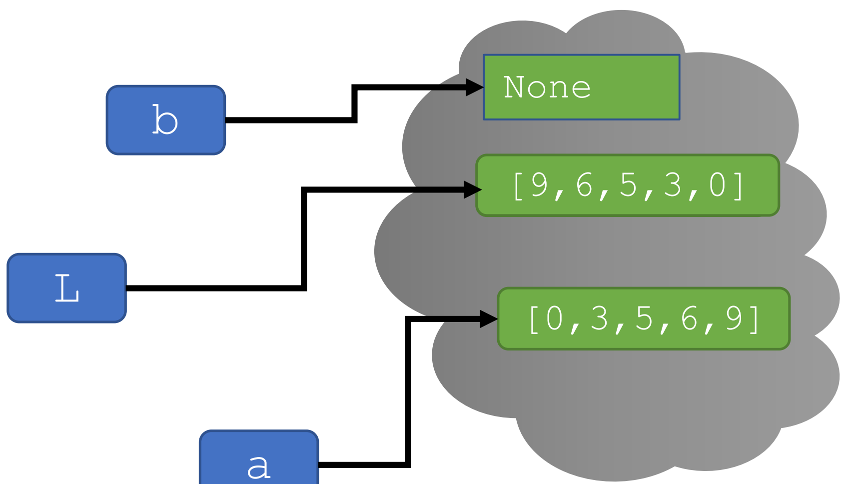
```
L=[9,6,0,3]
L.append(5)
```

`a = sorted(L)` → returns a **new** sorted list, does **not mutate** `L`

`b = L.sort()` → **mutates** `L` to be `[0,3,5,6,9]` and returns None

`L.reverse()` → **mutates** `L` to be `[9,6,5,3,0]` and returns None

Remember, we have to invoke the function even if it takes no arguments



YOU TRY IT!

- Write a function that meets these specs:

```
def sort_words(sen):
    """ sen is a string representing a sentence
    Returns a list containing all the words in sen but
    sorted in alphabetical order. """

print(sort_words("look at this photograph"))
```

BIG IDEA

Functions with side effects mutate inputs.

You can write your own!

LISTS SUPPORT ITERATION

- Let's write a **function that mutates the input**
- Example: square every element of a list, mutating original list

```
def square_list(L):
    for elem in L:
        # ?? How to do L[index] = the square ??
        # ?? elem is an element in L, not the index :(
```

- Solutions (we'll go over option 2, try the others on your own!):
  - Option 1: Make a **new variable** representing the index, initialized to 0 before the loop and incremented by 1 in the loop.
  - Option 2: **Loop over the index** not the element, and use L[index] to get the element
  - Option 3: Use **enumerate** in the for loop (I leave this option to you to look up). i.e. `for i,e in enumerate(L)`

LISTS SUPPORT ITERATION

- Example: square every element of a list, mutating original list

```
def square_list(L):
    for i in range(len(L)):
        L[i] = L[i]**2
```

To change elements of list, need to loop over indices into list

An assignment statement. L[i] is not a name, but points to a particular spot in the list data structure.

L[i] is the element

- Note, **no return**!

TRACE the CODE with an EXAMPLE

- Example: square every element of a list, mutating original list

```
def square_list(L):
    for i in range(len(L)):
        L[i] = L[i]**2
```

Suppose L is [2,3,4]

i is 0: L is mutated to [4, 3, 4]

i is 1: L is mutated to [4, 9, 4]

i is 2: L is mutated to [4, 9, 16]

TRACE the CODE with an EXAMPLE

- Example: square every element of a list, mutating original list

```
def square_list(L):
    for i in range(len(L)):
        L[i] = L[i]**2
```

The function mutates the input object passed in (Lin)

```
Lin = [2,3,4]
print("before fcn call:",Lin)   # prints [2,3,4]
square_list(Lin)
print("after fcn call:",Lin)    # prints [4,9,16]
```

No variable to assign function call to!

BIG IDEA

Functions that mutate the input likely.....

Iterate over len(L) not L.

Return None, so the function call does not need to be saved.

MUTATION

- Lists are **mutable** structures
- There are many advantages to being able to **change a portion** of a list
  - Suppose I have a very long list (e.g. of personnel records) and I want to update one element. Without mutation, I would have to copy the entire list, with a new version of that record in the right spot. A mutable structure lets me change just that element
- But, this ability can also introduce unexpected challenges

TRICKY EXAMPLES OVERVIEW

- TRICKY EXAMPLE 1:
  - A loop iterates over **indices of L** and **mutates L** each time (adds more elements).
- TRICKY EXAMPLE 2:
  - A loop iterates over **L's elements** directly and **mutates L** each time (adds more elements).
- TRICKY EXAMPLE 3:
  - A loop iterates over **L's elements** directly but **reassigns L** to a new object each time
- TRICKY EXAMPLE 4 (next time):
  - A loop iterates over **L's elements** directly and mutates L by **removing elements**.

TRICKY EXAMPLE 1: append

- **Range returns something that behaves like a tuple** (but isn't – it returns an *iterable*)
  - Returns the first element, and an iteration method by which subsequent elements are generated as needed

`range(4)` → *kind of like* tuple `(0,1,2,3)`

`range(2,9,2)` → *kind of like* tuple `(2,4,6,8)`

```
L = [1,2,3,4]
for i in range(len(L)):
    L.append(i)
print(L)
```

Iteration sequence is pre-determined at beginning of loop

1st time: L is [1, 2, 3, 4, 0]

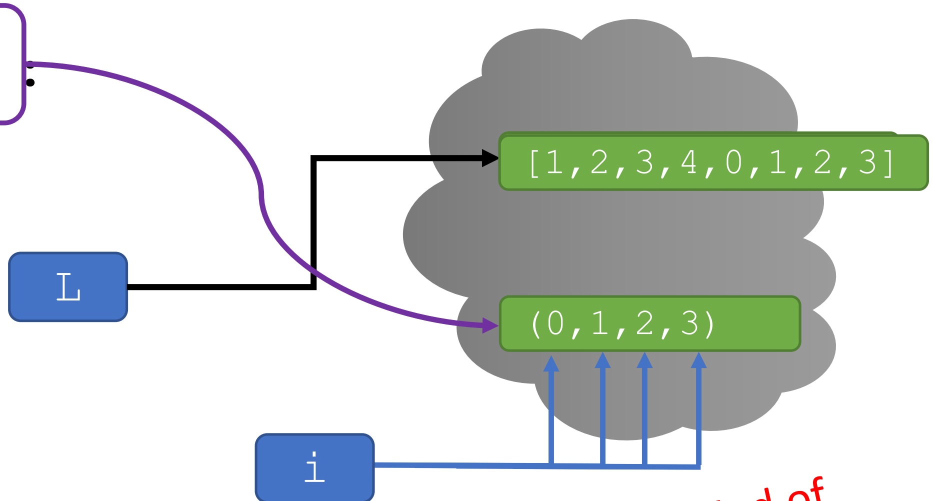
2nd time: L is [1, 2, 3, 4, 0, 1]

3rd time: L is [1, 2, 3, 4, 0, 1, 2]

4th time: L is [1, 2, 3, 4, 0, 1, 2, 3]

TRICKY EXAMPLE 1: append

```
L = [1,2,3,4]
for i in range(len(L)):
    L.append(i)
    print(L)
```



1st time: L is [1, 2, 3, 4, 0]

2nd time: L is [1, 2, 3, 4, 0, 1]

3rd time: L is [1, 2, 3, 4, 0, 1, 2]

4th time: L is [1, 2, 3, 4, 0, 1, 2, 3]

`i` iterates over a "tuple" created by range; mutation of `L` does not affect this "tuple"

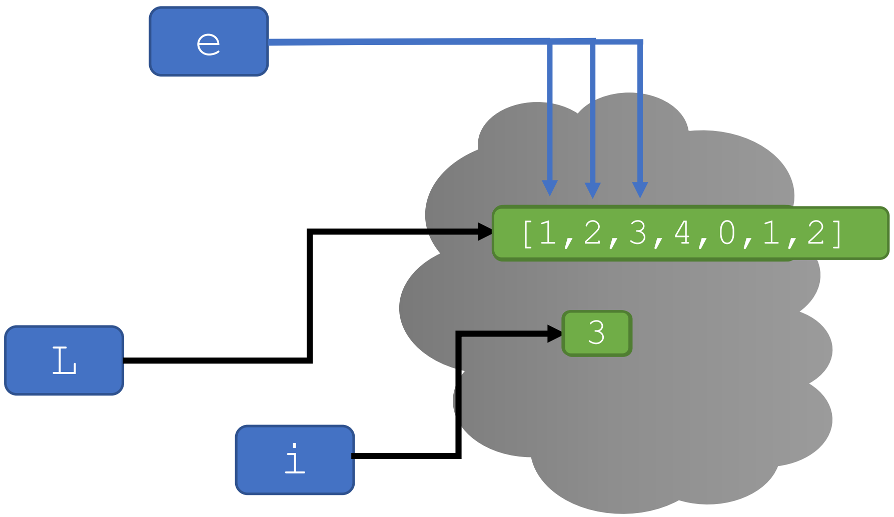
TRICKY EXAMPLE 2: append

Looks similar **but** ...

```
L = [1,2,3,4]
i = 0
for e in L:
    L.append(i)
    i += 1
    print(L)
```

Originally [1,2,3,4]

L is **mutated** each iteration



In previous example, L was accessed at onset to create a range iterable; in this example, the loop is directly accessing indices into L

1st time: L is [1, 2, 3, 4, 0]

2nd time: L is [1, 2, 3, 4, 0, 1]

3rd time: L is [1, 2, 3, 4, 0, 1, 2]

4th time: L is [1, 2, 3, 4, 0, 1, 2, 3]

NEVER STOPS!

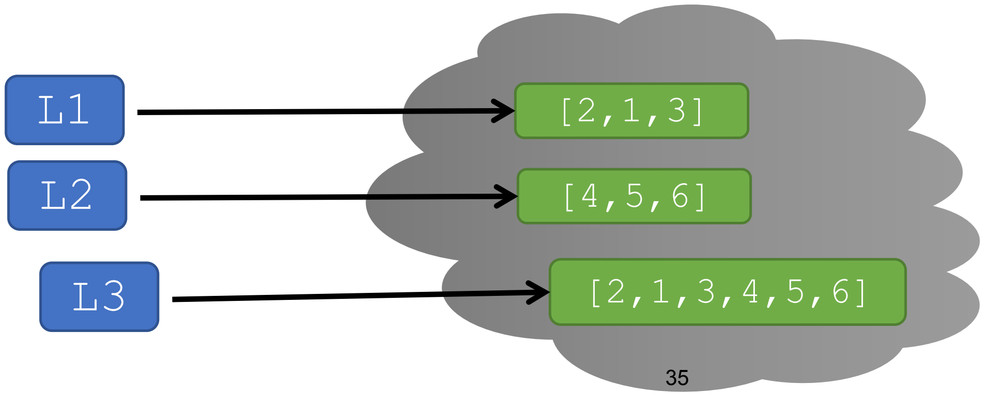
COMBINING LISTS

Remember strings

- **Concatenation**, + operator, creates a **new** list, with copies
- **Mutate** list with `L.extend(some_list)` (copy of `some_list`)

```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2
```

→ `L3` is `[2,1,3,4,5,6]`



concatenation creates **new** list with copies

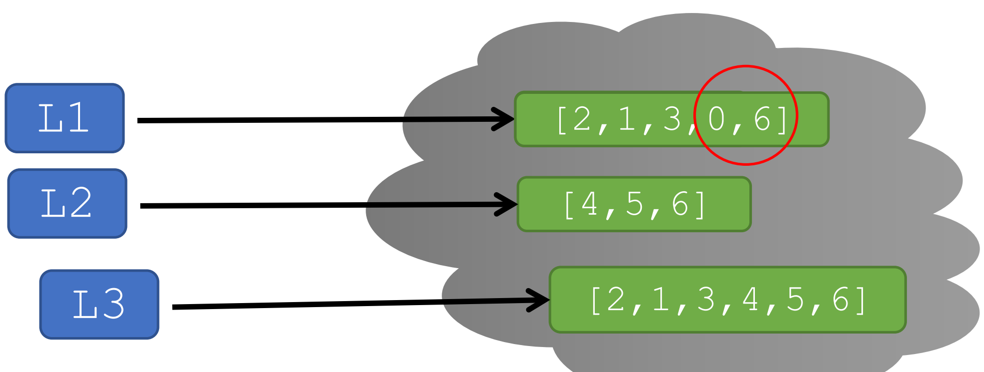
COMBINING LISTS

- **Concatenation**, + operator, creates a **new** list, with copies
- **Mutate** list with `L.extend(some_list)` (copy of `some_list`)

```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2
L1.extend([0,6])
```

→ `L3` is `[2,1,3,4,5,6]`

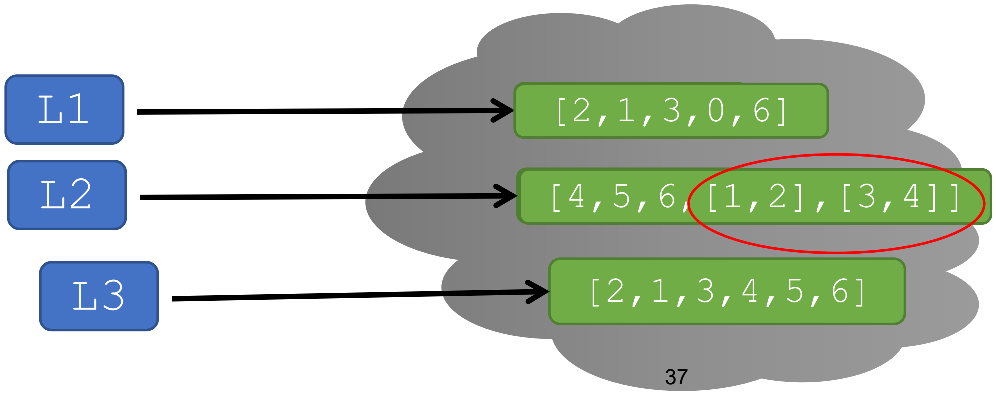
→ mutate `L1` to `[2,1,3,0,6]`



COMBINING LISTS

- **Concatenation**, + operator, creates a **new** list, with copies
- **Mutate** list with `L.extend(some_list)` (copy of `some_list`)

```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2                 → L3 is [2,1,3,4,5,6]
L1.extend([0,6])             → mutate L1 to [2,1,3,0,6]
L2.extend([[1,2],[3,4]])     → mutates L2 to [4,5,6,[1,2],[3,4]]
```



Extending by a list of lists gives us new list elements

TRICKY EXAMPLE 3: combining

```
L = [1,2,3,4]
for e in L:
    L = L + L
    print(L)
```

Originally [1,2,3,4]

L is **bound to a new object** each iteration; but looping of e walks down structure pointed to when called, so iterates only 4 times, over original [1,2,3,4]

1st time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4]

2nd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

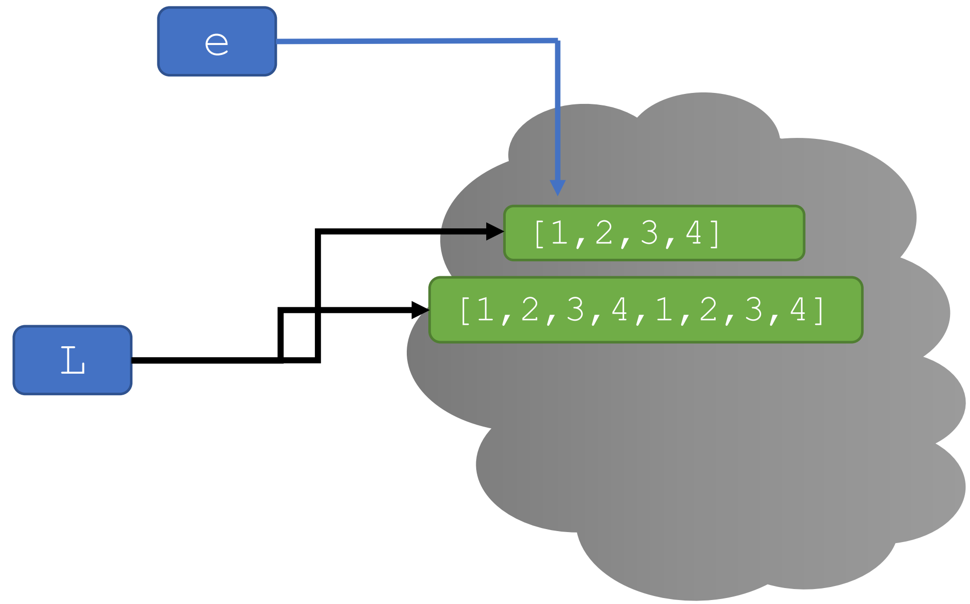
3rd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

4th time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

TRICKY EXAMPLE 3: combining

Note: e is still indexing into original data structure

```
L = [1,2,3,4]
for e in L:
    L = L + L
    print(L)
```

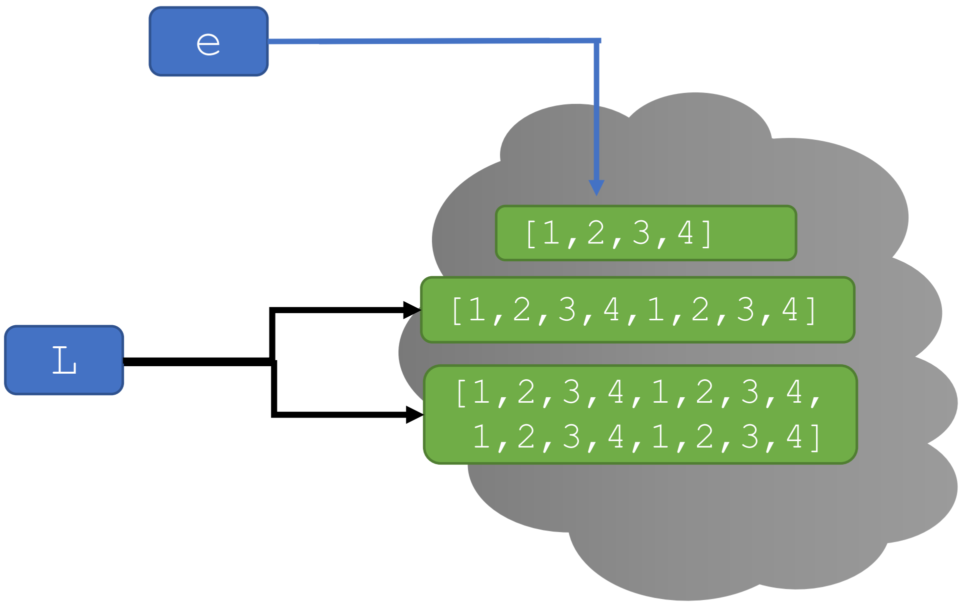


1st time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4]

TRICKY EXAMPLE 3: combining

Note: e is still indexing into original data structure

```
L = [1,2,3,4]
for e in L:
    L = L + L
    print(L)
```



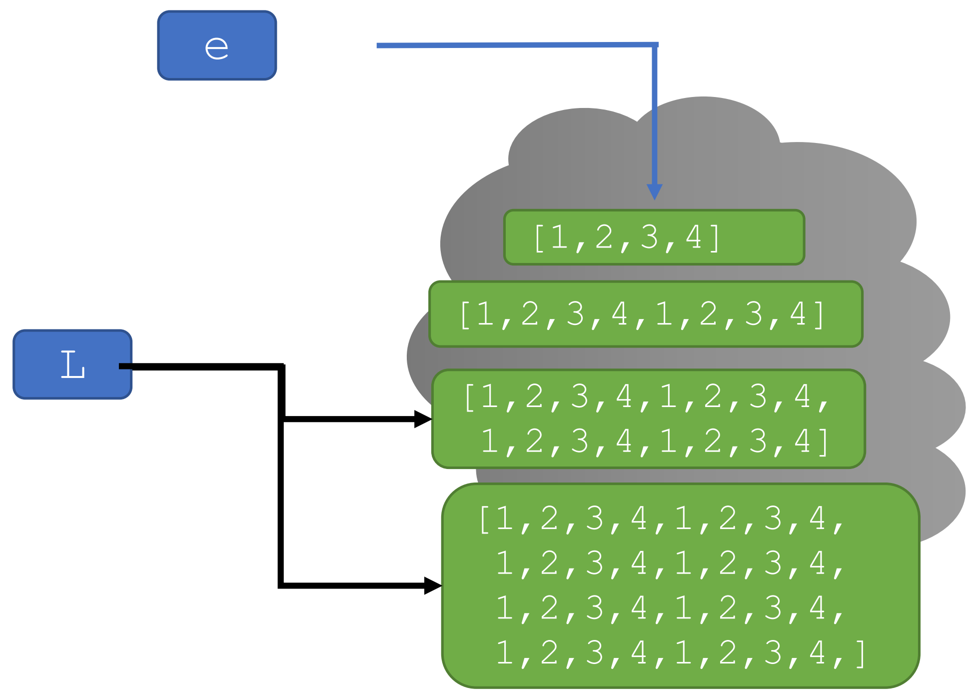
1st time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4]

2nd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

TRICKY EXAMPLE 3: combining

Note: e is still indexing into original data structure

```
L = [1,2,3,4]
for e in L:
    L = L + L
    print(L)
```



1st time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4]

2nd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

3rd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4 ,
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

TRICKY EXAMPLE 3: combining

Note: e is still indexing into original data structure

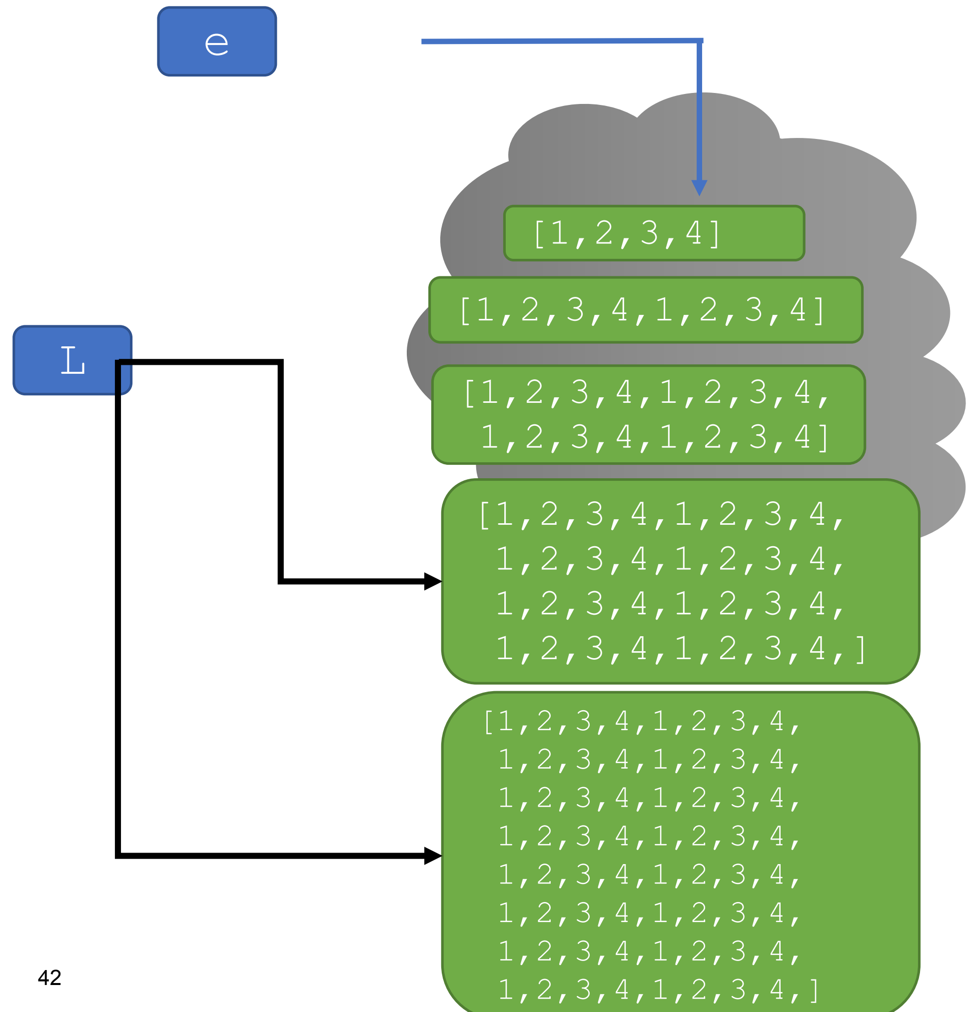
```
L = [1,2,3,4]

for e in L:

    L = L + L

    print(L)
```

4th time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4 ,
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4 ,
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]



EMPTY OUT A LIST AND CHECKING THAT IT'S THE SAME OBJECT

- You can **mutate a list to remove all its elements**
  - This **does not make a new empty list**!
- Use `L.clear()`
- How to check that it's **the same object in memory**?
  - Use the id() function
  - Try this in the console

```
>>> L = [4,5,6]
>>> id(L)
>>> L.append(8)
>>> id(L)
>>> L.clear()
>>> id(L)
```

Same!

```
>>> L = [4,5,6]
>>> id(L)
>>> L.append(8)
>>> id(L)
>>> L = []
>>> id(L)
```

Different!

SUMMARY

- Lists and tuples provide a way to organize data that naturally supports iterative functions
- Tuples are **immutable** (like strings)
  - Tuples are useful when you have **data that doesn't need to change**.
    e.g. (latitude, longitude) or (page #, line #)
- Lists are **mutable**
  - You can modify the object by **changing an element** at an index
  - You can modify the object by **adding elements** to the end
  - Will see many more operations on lists next time
  - Lists are useful in **dynamic situations**.
    e.g. a list of daily top 40 songs or a list of recently watched movies