Lecture 12: Aliasing and Cloning

Comp 102

Fakhir Shaheen

Forman Christian University

Assignment on Lists

```
1 Loriginal = [4, 5, 6]
2 Lnew = Loriginal  # Different name, SAME object
3 Lnew[0] = 10
4
5 print(Loriginal)  # Output: [10, 5, 6]
6 print(Lnew)       # Output: [10, 5, 6]
```

- *Note:* `Lnew = Loriginal` does NOT create a copy. Both names refer to the same list object.

Assignment on Lists

```
1 Loriginal = [4, 5, 6]
2 Lnew = Loriginal  # Different name, SAME object
3 Lnew[0] = 10
4
5 print(Loriginal)  # Output: [10, 5, 6]
6 print(Lnew)       # Output: [10, 5, 6]
```

- *Note:* `Lnew = Loriginal` does NOT create a copy. Both names refer to the same list object.
- `Lnew` is an **alias** of `Loriginal`: a second name for the same object.
- A real copy, a second object with the same values, is called a **clone**.

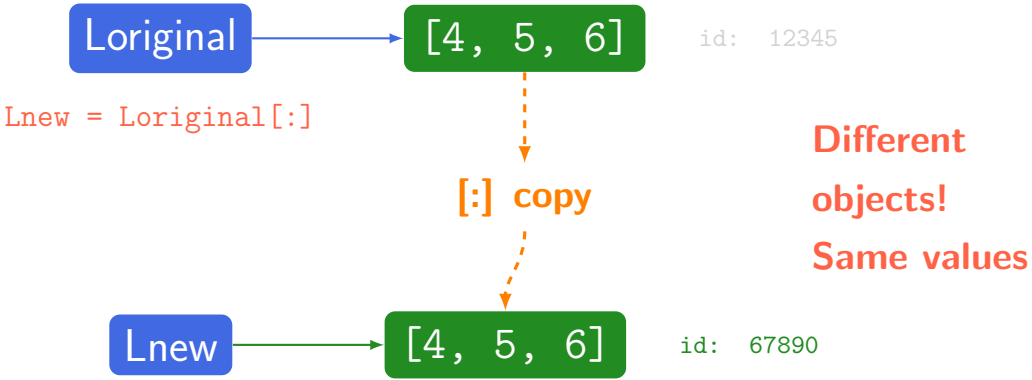
Creating Copies of Lists

- Assignment operator (=) **Does Not** create a copy

Creating Copies of Lists

- Assignment operator (=) **Does Not** create a copy
- To create a copy of a list, we can use:
  - Slicing: `new_list = old_list[:]`
  - The `list()` function: `new_list = list(old_list)`
  - The `copy()` method: `new_list = old_list.copy()`

Creating Copies of Lists



Slicing creates a shallow copy - a new list object

Shallow Copy Example

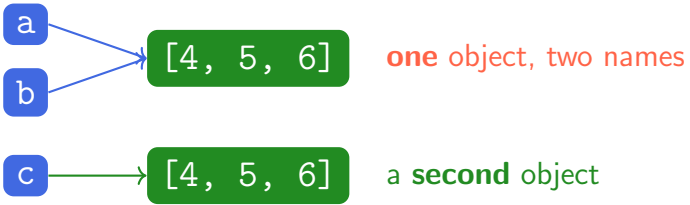
```
1 Loriginal = [4, 5, 6]
2 Lnew = Loriginal[:]  # or Loriginal.copy()
3 Lnew[0] = 10
4
5 print(Loriginal)  # Output: [4, 5, 6]
6 print(Lnew)       # Output: [10, 5, 6]
```

- *Note:* `Lnew = Loriginal[:]` creates a copy. `Lnew` and `Loriginal` refer to different list objects.

One List or Two?

Which of b and c is the **same list** as a? You can ask Python.

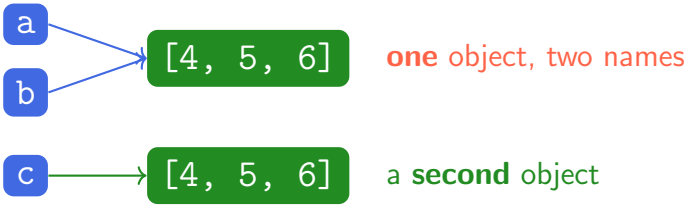
```
1 a = [4, 5, 6]
2 b = a          # alias
3 c = a[:]       # clone
```



One List or Two?

Which of b and c is the **same list** as a? You can ask Python.

```
1  a = [4, 5, 6]
2  b = a          # alias
3  c = a[:]       # clone
```



== asks: same **contents**?

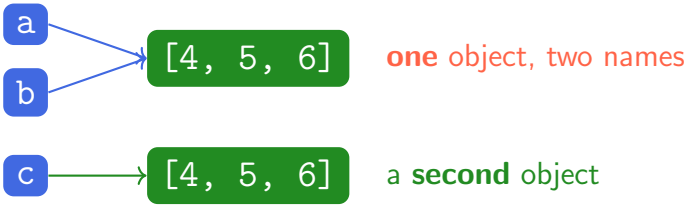
a == b → True

a == c → True

One List or Two?

Which of `b` and `c` is the **same list** as `a`? You can ask Python.

```
1  a = [4, 5, 6]
2  b = a          # alias
3  c = a[:]       # clone
```



`==` asks: same **contents**?

`a == b` → `True`

`a == c` → `True`

`is` asks: same **object**? (same `id()`, Lecture 11)

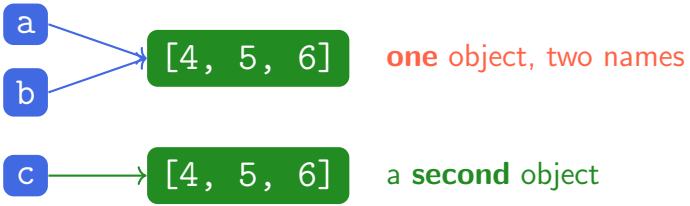
`a is b` → `True`

`a is c` → `False`

One List or Two?

Which of `b` and `c` is the **same list** as `a`? You can ask Python.

```
1  a = [4, 5, 6]
2  b = a          # alias
3  c = a[:]       # clone
```



`==` asks: same **contents**?

- `a == b` → `True`
- `a == c` → `True`

`is` asks: same **object**? (same `id()`, Lecture 11)

- `a is b` → `True`
- `a is c` → `False`

So: `==` cannot tell an alias from a clone. `is` can.

You Try!

- Write a function that meets the specification below:

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None
    """
    pass

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

You Try!

- Write a function that meets the specification below:
- *Hint:* Make a copy to save the elements. Then use `L.clear()` to empty out the list and repopulate it with the ones you're keeping.

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None
    """
    pass

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

Solution

```
1  def remove_all(L, e):
2      copy = L[:]                # Make a copy of the original list
3      L.clear()                  # Clear the original list
4      for item in copy:          # Iterate over the copy
5          if item != e:          # If the item is not equal to e
6              L.append(item)     # Append it back to the original list
7      return None
8
9
10 L = [1,2,2,2,3,4,2]
11 remove_all(L, 2)
12 print(L) # prints [1, 3, 4]
```

Deleting Items from Lists

- `del L[i]`: Deletes the item at index `i` from list `L`. Example:

```
L = [10, 20, 30, 40]
del L[1]  # Removes the item at index 1 (20)
print(L)  # Output: [10, 30, 40]
```

Deleting Items from Lists

- `del L[i]`: Deletes the item at index `i` from list `L`. Example:

```
L = [10, 20, 30, 40]
del L[1]  # Removes the item at index 1 (20)
print(L)  # Output: [10, 30, 40]
```

- `L.pop(i)`: Removes and returns the item at index `i` from list `L`. If no index is specified, it removes and returns the last item. Example:

```
L = [10, 20, 30, 40]
item = L.pop(1)  # Removes and returns item at index 1 (20)
print(item)      # Output: 20
print(L)         # Output: [10, 30, 40]
```

Deleting Items from Lists

- `L.remove(x)`: Removes the first occurrence of value `x` from list `L`. Gives error if `x` is not found.

```
L = [10, 20, 30, 20, 40]
L.remove(20)  # Removes the FIRST occurrence of 20
print(L)      # Output: [10, 30, 20, 40]
```

Deleting Items from Lists

- `L.remove(x)`: Removes the first occurrence of value `x` from list `L`. Gives error if `x` is not found.

```
L = [10, 20, 30, 20, 40]
L.remove(20)  # Removes the FIRST occurrence of 20
print(L)      # Output: [10, 30, 20, 40]
```

- all of these `del`, `pop()`, and `remove()` modify the list in place and return `None` *(except `pop()` which returns the removed item)*.

Rewriting above code using `remove()`

```
1  def remove_all(L, e):
2      """
3      L is a list
4      Mutates L to remove all elements in L that are equal to e
5      Returns None.
6      """
7      while e in L:
8          L.remove(e)
```

Note: This is less efficient than the previous solution since each call to `remove()` searches the list for `e`.

What if the code was this:

```
def remove_all(L, e):
    """
    L is a list
    Mutates L to remove all elements in L that are equal to e
    Returns None.
    """
    for item in L:
        if item == e:
            L.remove(item)
    return None

L = [1,2,2,2]
remove_all(L, 2)
print(L) # Should print [1]
```

What if the code was this:

```
1  def remove_all(L, e):
2      """
3      L is a list
4      Mutates L to remove all elements in L that are equal to e
5      Returns None.
6      """
7      for item in L:
8          if item == e:
9              L.remove(item)
10     return None
11
12 L = [1,2,2,2]
13 remove_all(L, 2)
14 print(L) # Should print [1]
```

-> Actually prints [1, 2]

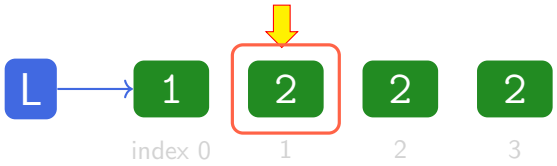
Tricky: Mutating While Iterating

```
1  def remove_all(L, e):
2    for item in L:
3      if item == e:
4        L.remove(item)
5
6  L = [1, 2, 2, 2]
7  remove_all(L, 2)
```



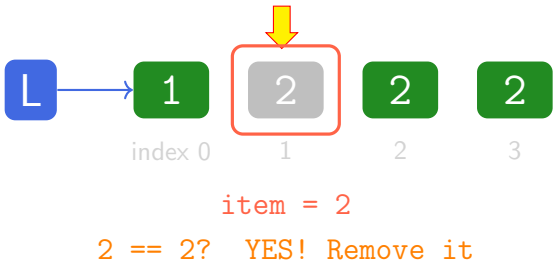
Tricky: Mutating While Iterating

```
1  def remove_all(L, e):
2    for item in L:
3      if item == e:
4        L.remove(item)
5
6  L = [1, 2, 2, 2]
7  remove_all(L, 2)
```



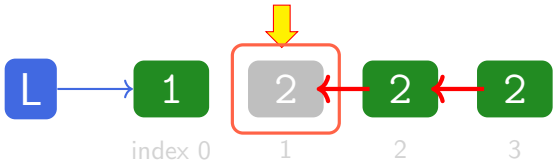
Tricky: Mutating While Iterating

```
1  def remove_all(L, e):
2    for item in L:
3      if item == e:
4        L.remove(item)
5
6  L = [1, 2, 2, 2]
7  remove_all(L, 2)
```



Tricky: Mutating While Iterating

```
1  def remove_all(L, e):
2    for item in L:
3      if item == e:
4        L.remove(item)
5
6  L = [1, 2, 2, 2]
7  remove_all(L, 2)
```



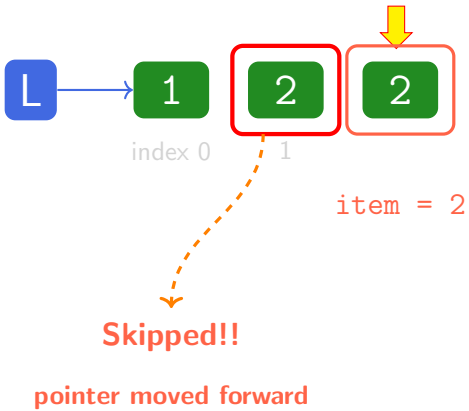
Tricky: Mutating While Iterating

```
1  def remove_all(L, e):
2    for item in L:
3      if item == e:
4        L.remove(item)
5
6  L = [1, 2, 2, 2]
7  remove_all(L, 2)
```



Tricky: Mutating While Iterating

```
1  def remove_all(L, e):
2    for item in L:
3      if item == e:
4        L.remove(item)
5
6  L = [1, 2, 2, 2]
7  remove_all(L, 2)
```



Tricky: Mutating While Iterating

```
1  def remove_all(L, e):
2    for item in L:
3      if item == e:
4        L.remove(item)
5
6  L = [1, 2, 2, 2]
7  remove_all(L, 2)
```



Result: [1, 2]

One **2** remains!

Expected: [1]

You Try!

Want to mutate L1 to remove any elements that are also in L2.
Try running in pythontutor.

```
1  def remove_dups(L1, L2):
2      for e in L1:
3          if e in L2:
4              L1.remove(e)
5
6  L1 = [10, 20, 30, 40]
7  L2 = [10, 20, 50, 60]
8  remove_dups(L1, L2)
9  print(L1) # expected:[30,40]
```

You Try!

Want to mutate L1 to remove any elements that are also in L2.
Try running in pythontutor.

```
1  def remove_dups(L1, L2):
2      for e in L1:
3          if e in L2:
4              L1.remove(e)
5
6  L1 = [10, 20, 30, 40]
7  L2 = [10, 20, 50, 60]
8  remove_dups(L1, L2)
9  print(L1) # expected:[30,40]
```

-> Actual:[20,30,40]

Mutation and iteration without clone




```
e = 10
10 in L2?  Yes, delete
```

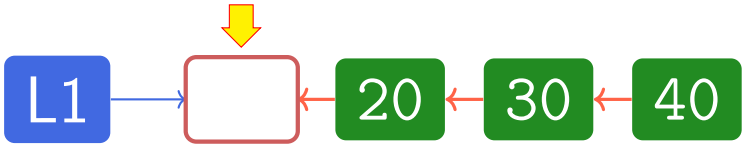
```
L2 = [10,20,50,60]
```

Mutation and iteration without clone




```
L2 = [10,20,50,60]
```

Mutation and iteration without clone



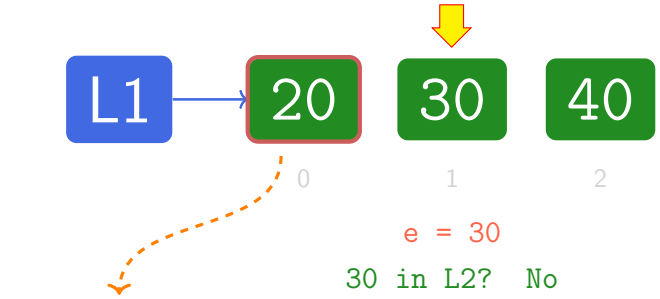

```
L2 = [10,20,50,60]
```

Mutation and iteration without clone




```
L2 = [10,20,50,60]
```

Mutation and iteration without clone



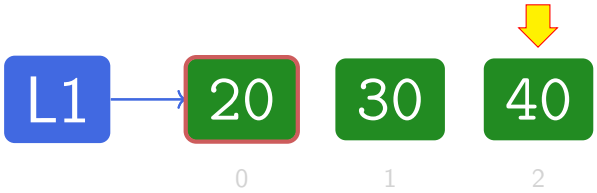

```
L2 = [10,20,50,60]
```

Mutation and iteration without clone



L2 = [10,20,50,60]

Mutation and iteration without clone



e = 40

40 in L2? No

Result: [20,30,40]

Expected: [30,40]

```
L2 = [10,20,50,60]
```

Solution

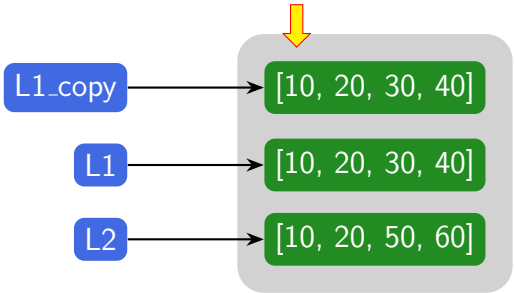
```
1  def remove_dups(L1, L2):
2      L1_copy = L1[:]          # make a copy of L1
3      for e in L1_copy:        # iterate over a COPY of L1
4          if e in L2:
5              L1.remove(e)     # remove L1, not the copy
6
7  L1 = [10, 20, 30, 40]
8  L2 = [10, 20, 50, 60]
9  remove_dups(L1, L2)
10 print(L1) # expected:[30,40]
```

Solution

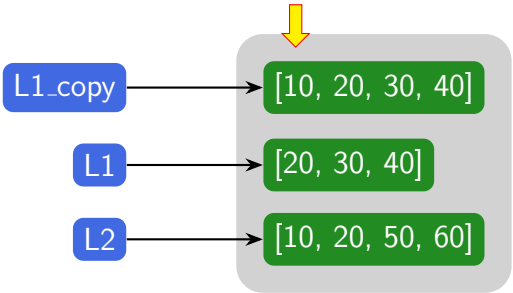
```
1  def remove_dups(L1, L2):
2      L1_copy = L1[:]          # make a copy of L1
3      for e in L1_copy:        # iterate over a COPY of L1
4          if e in L2:
5              L1.remove(e)     # remove L1, not the copy
6
7  L1 = [10, 20, 30, 40]
8  L2 = [10, 20, 50, 60]
9  remove_dups(L1, L2)
10 print(L1) # expected:[30,40]
```

-> Actual:[30,40]

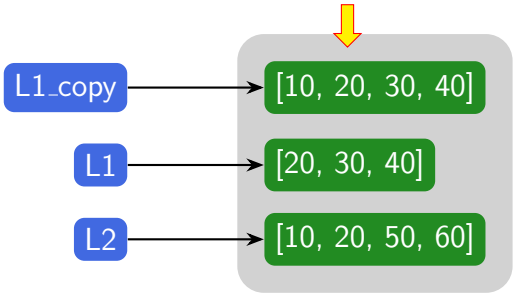
Solution Visualization



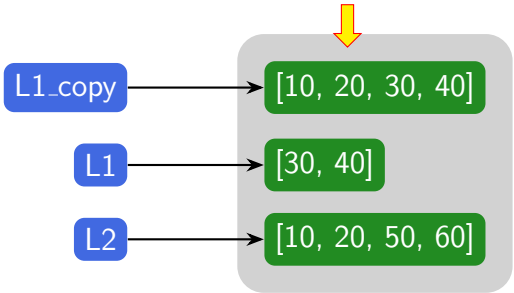
Solution Visualization



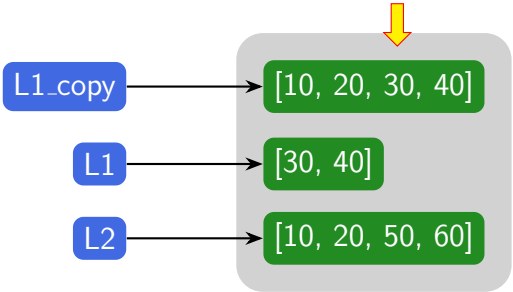
Solution Visualization



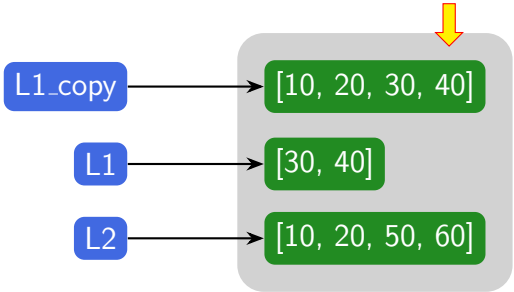
Solution Visualization



Solution Visualization



Solution Visualization



Big Idea

Never add or **delete** items from a list while **iterating** over it.

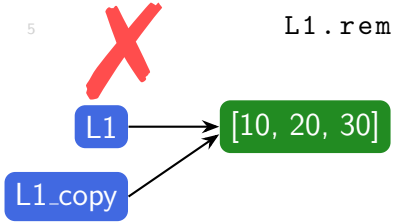
Mutation and Iteration with Alias

```
def remove_dups(L1, L2):
    L1_copy = L1
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

Mutation and Iteration with Alias

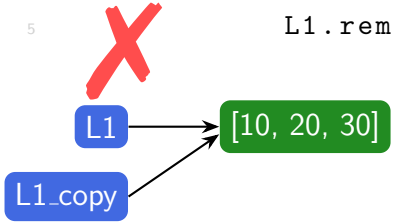
```
1 def remove_dups(L1, L2):
2     L1_copy = L1
3     for e in L1_copy:
4         if e in L2:
5             L1.remove(e)
```



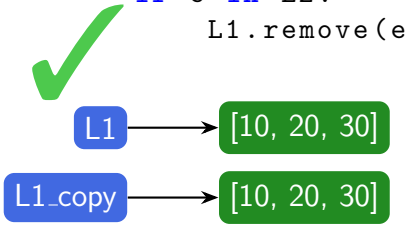

```
1 def remove_dups(L1, L2):
2     L1_copy = L1[:]
3     for e in L1_copy:
4         if e in L2:
5             L1.remove(e)
```

Mutation and Iteration with Alias

```
1 def remove_dups(L1, L2):
2     L1_copy = L1
3     for e in L1_copy:
4         if e in L2:
5             L1.remove(e)
```

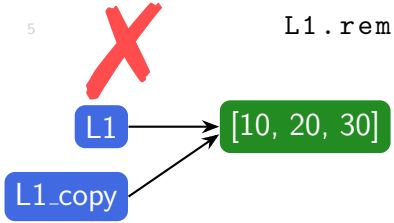



```
1 def remove_dups(L1, L2):
2     L1_copy = L1[:]
3     for e in L1_copy:
4         if e in L2:
5             L1.remove(e)
```

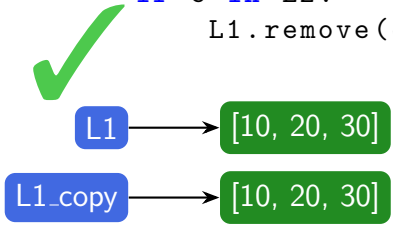


Mutation and Iteration with Alias

```
def remove_dups(L1, L2):
    L1_copy = L1
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```




```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```



Simple assignment (=): **Does Not** create a copy

Big Idea

Assignment (=) **Does Not** create a copy. It creates an **Alias**

Use slicing `[:]`, `list()`, or `copy()` to create a copy.

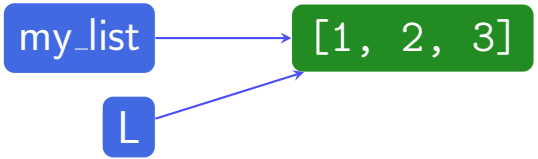
Lists as Parameters

- Lists are mutable, so when passed as parameters, functions can modify the original list. Example:

Lists as Parameters

- Lists are mutable, so when passed as parameters, functions can modify the original list. Example:

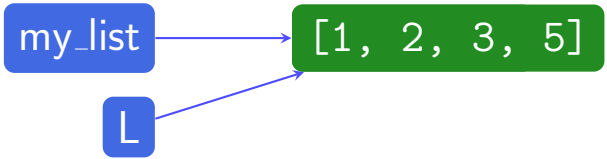
```
1 def append_five(L):
2     L.append(5)
3
4 my_list = [1, 2, 3]
5 append_five(my_list)
6 print(my_list)  # Output: [1, 2, 3, 5]
```



Lists as Parameters

- Lists are mutable, so when passed as parameters, functions can modify the original list. Example:

```
1 def append_five(L):
2     L.append(5)
3
4 my_list = [1, 2, 3]
5 append_five(my_list)
6 print(my_list)  # Output: [1, 2, 3, 5]
```

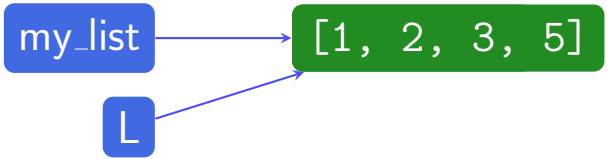


Lists as Parameters

- Lists are mutable, so when passed as parameters, functions can modify the original list. Example:

```
1 def append_five(L):
2     L.append(5)
3
4 my_list = [1, 2, 3]
5 append_five(my_list)
6 print(my_list)  # Output: [1, 2, 3, 5]
```

my_list and L are **Aliases**



You Try!

On paper: **(1)** In one sentence, what does `lowest_two` return?
(2) What do the two `print` lines show?

```
1  def lowest_two(marks):
2      marks.sort()
3      return marks[0:2]
4
5  quiz = [7, 10, 4, 9]
6  print(lowest_two(quiz))
7  print(quiz)
```

You Try!

On paper: **(1)** In one sentence, what does `lowest_two` return?
(2) What do the two `print` lines show?

```
1  def lowest_two(marks):
2      marks.sort()
3      return marks[0:2]
4
5  quiz = [7, 10, 4, 9]
6  print(lowest_two(quiz)) → [4, 7]
7  print(quiz) → [4, 7, 9, 10]
```

You Try!

On paper: **(1)** In one sentence, what does `lowest_two` return?
(2) What do the two `print` lines show?

```
1  def lowest_two(marks):
2      marks.sort()
3      return marks[0:2]
4
5  quiz = [7, 10, 4, 9]
6  print(lowest_two(quiz)) → [4, 7]
7  print(quiz) → [4, 7, 9, 10]
```

✓ Returns the two lowest marks.

✗ It also sorts the caller's list! `marks` is an **alias** of `quiz`, and `sort()` mutates.

You Try!

On paper: **(1)** In one sentence, what does `lowest_two` return?
(2) What do the two `print` lines show?

```
1  def lowest_two(marks):
2      marks.sort()
3      return marks[0:2]
4
5  quiz = [7, 10, 4, 9]
6  print(lowest_two(quiz)) → [4, 7]
7  print(quiz) → [4, 7, 9, 10]
```

✓ Returns the two lowest marks.

✗ It also sorts the caller's list! `marks` is an **alias** of `quiz`, and `sort()` mutates.

Fix: work on a clone.

```
ordered = marks[:]
ordered.sort()
return ordered[0:2]
```

Now `quiz` stays `[7, 10, 4, 9]`.
(Or: `ordered = sorted(marks)`.)

Nested List Structure

- Lists can contain other lists as elements, creating a nested structure.
- Example:

```
nested_list = [[1, 2], [3, 4]]
print(nested_list[0])     # Output: [1, 2]
print(nested_list[1][1])  # Output: 4
```

Nested List Structure

- Lists can contain other lists as elements, creating a nested structure.
- Example:

```
nested_list = [[1, 2], [3, 4]]
print(nested_list[0])     # Output: [1, 2]
print(nested_list[1][1])  # Output: 4
```

- You can modify elements in nested lists using indexing.

```
nested_list[0][1] = 20
print(nested_list)  # Output: [[1, 20], [3, 4]]
```

What Is Inside a List?

```
1 L = [7, 4, 9] ←
2 print(L[1])
```



Python makes the objects. Each one has an **id**: where it lives.

What Is Inside a List?

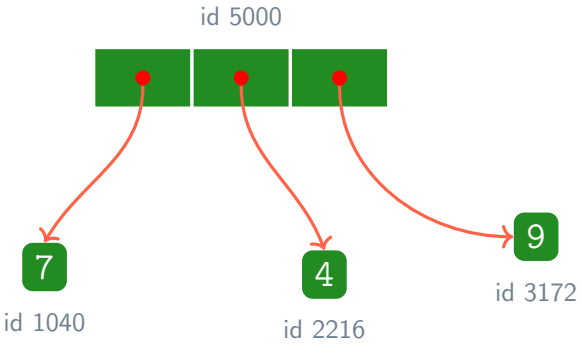
```
1 L = [7, 4, 9] ←
2 print(L[1])
```



The list keeps their **ids**, not the numbers.

What Is Inside a List?

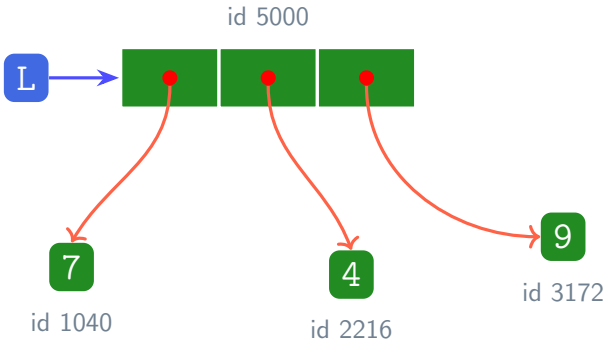
```
1 L = [7, 4, 9] ←
2 print(L[1])
```



We draw each id as an arrow: a **reference**.

What Is Inside a List?

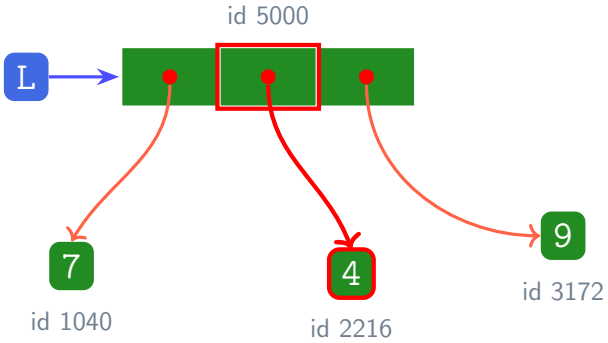
```
1 L = [7, 4, 9]
2 print(L[1])
```



A name is a reference too: `L` holds the list's id, 5000.

What Is Inside a List?

```
1 L = [7, 4, 9]
2 print(L[1]) # 4
```



`L[1]`: go to slot 1, follow its arrow.

One 2, Three Arrows

```
1 L = [2, 2, 2] ←
2 L[0] = 3
3 print(L)
```

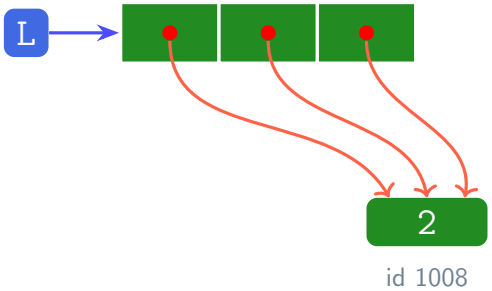




Three slots, the **same** id: there is only **one** 2.

One 2, Three Arrows

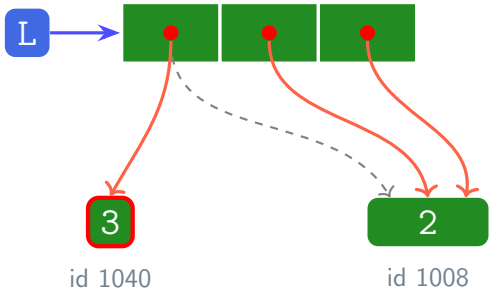
```
1 L = [2, 2, 2]
2 L[0] = 3
3 print(L)
```



Python shares its one 2. Safe: a number can never change.

One 2, Three Arrows

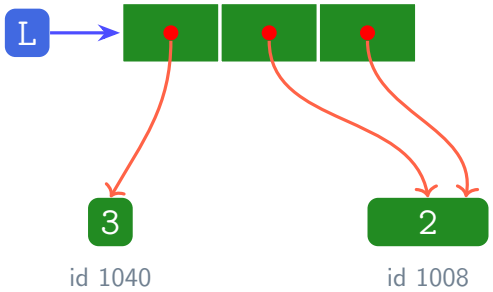
```
1 L = [2, 2, 2]
2 L[0] = 3 ←
3 print(L)
```



Slot 0 gets a new arrow. The 2 is untouched.

One 2, Three Arrows

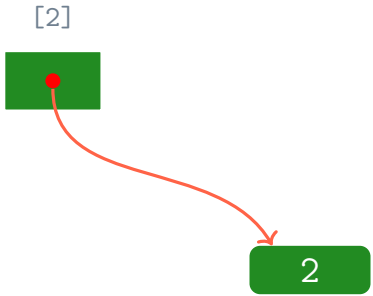
```
1 L = [2, 2, 2]
2 L[0] = 3
3 print(L) # [3, 2, 2]
```



`print` follows each arrow.

* Copies the Arrow

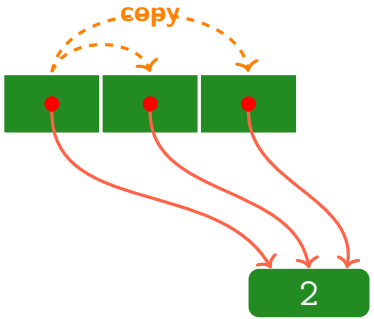
```
1 A = [2] * 3
```



[2]: one slot, one arrow to the 2.

* Copies the Arrow

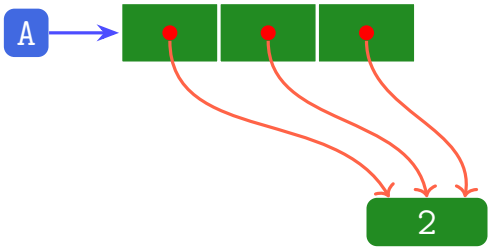
```
1 A = [2] * 3
```



* 3 copies the **arrow**, not the 2.

* Copies the Arrow

```
1 A = [2] * 3
```



The same picture as `[2, 2, 2]`.

Why Arrows Matter

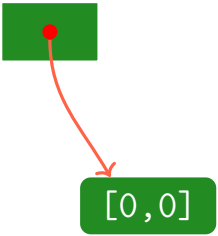
```
1 M = [[0] * 2] * 3 ←
2 M[0][0] = 5
3 print(M)
```

[0,0]

[0] * 2 makes **one** row.

Why Arrows Matter

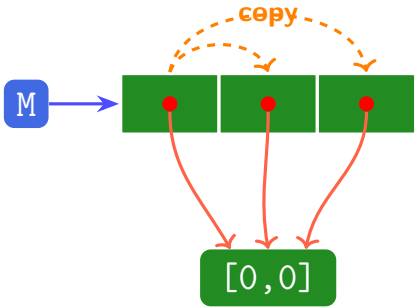
```
1 M = [[0] * 2] * 3
2 M[0][0] = 5
3 print(M)
```



The outer [ ] holds one arrow to that row.

Why Arrows Matter

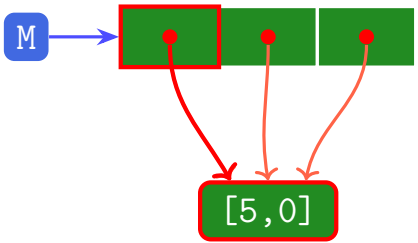
```
1 M = [[0] * 2] * 3
2 M[0][0] = 5
3 print(M)
```



`*` 3 copies the arrow: three arrows, **one** row.

Why Arrows Matter

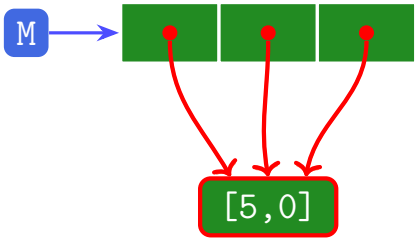
```
1 M = [[0] * 2] * 3
2 M[0][0] = 5 ←
3 print(M)
```



Follow slot 0's arrow, change the row.

Why Arrows Matter

```
1 M = [[0] * 2] * 3
2 M[0][0] = 5
3 print(M) ←
```

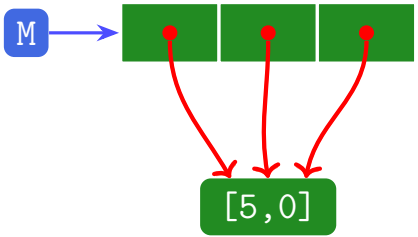



```
[[5, 0], [5, 0], [5, 0]]
```

All three arrows reach that **same** row.

Why Arrows Matter

```
1 M = [[0] * 2] * 3
2 M[0][0] = 5
3 print(M) ←
```




```
[[5, 0], [5, 0], [5, 0]]
```

Why we care: a shared 2 never changes. A shared list **can**.

Three Rows Need Three Lists

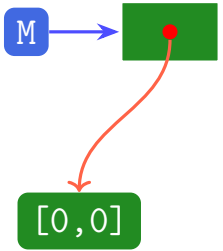
```
1 M = []
2 for i in range(3):
3     M.append([0] * 2)
4 M[0][0] = 5
5 print(M)
```



Start with an empty list.

Three Rows Need Three Lists

```
1 M = []
2 for i in range(3):
3     M.append([0] * 2)
4 M[0][0] = 5
5 print(M)
```

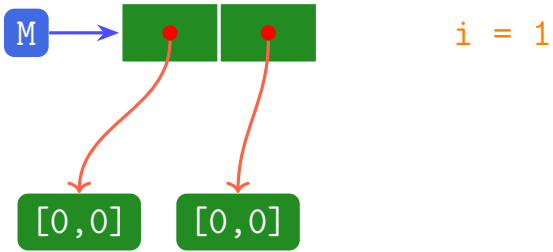


i = 0

Each pass makes a **new** `[0,0]`.

Three Rows Need Three Lists

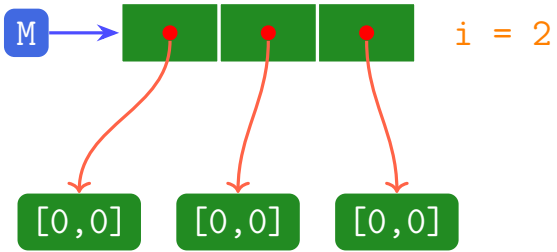
```
1 M = []
2 for i in range(3):
3     M.append([0] * 2)
4 M[0][0] = 5
5 print(M)
```



Each pass makes a **new** `[0,0]`.

Three Rows Need Three Lists

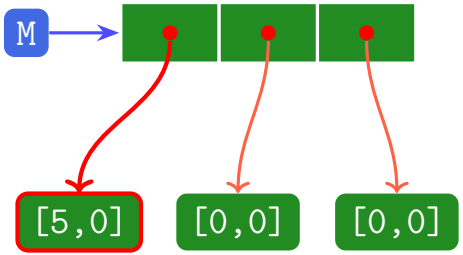
```
1 M = []
2 for i in range(3):
3     M.append([0] * 2)
4 M[0][0] = 5
5 print(M)
```



Each pass makes a **new** `[0,0]`.

Three Rows Need Three Lists

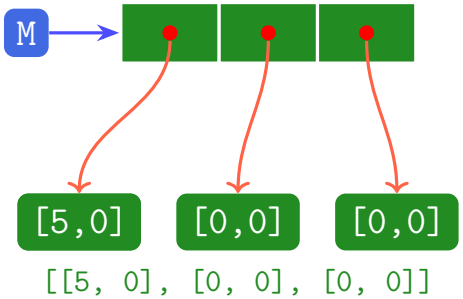
```
1 M = []
2 for i in range(3):
3     M.append([0] * 2)
4 M[0][0] = 5
5 print(M)
```



Three arrows, three rows: only the first one changes.

Three Rows Need Three Lists

```
1 M = []
2 for i in range(3):
3     M.append([0] * 2)
4 M[0][0] = 5
5 print(M) ←
```

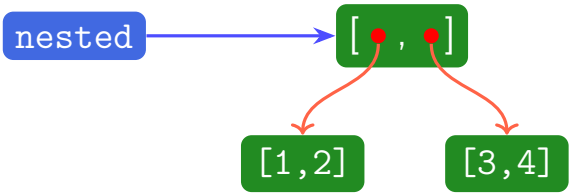


✓ One row changed, as we meant.

Nested Lists and Aliasing

```
1 nested = [[1, 2], [3, 4]]
2 L = nested
3 L[0][1] = 5
4 print(nested)
5 L[0] = [5, 6]
```

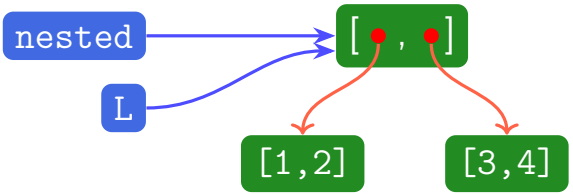
After: `nested = [[1, 2], [3, 4]]`



Nested Lists and Aliasing

```
1 nested = [[1, 2], [3, 4]]
2 L = nested
3 L[0][1] = 5
4 print(nested)
5 L[0] = [5, 6]
```

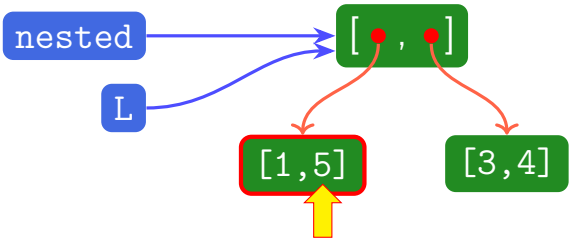
After: `L = nested` **(aliases!)**



Nested Lists and Aliasing

```
1 nested = [[1, 2], [3, 4]]
2 L = nested
3 L[0][1] = 5
4 print(nested)
5 L[0] = [5, 6]
```

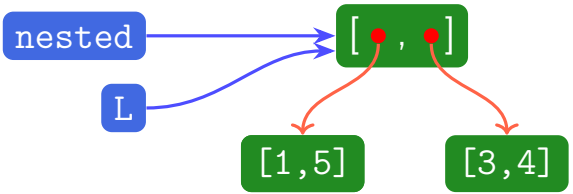
After: `L[0][1] = 5`



Nested Lists and Aliasing

```
1 nested = [[1, 2], [3, 4]]
2 L = nested
3 L[0][1] = 5
4 print(nested)  # Output:  [[1, 5], [3, 4]] ←
5 L[0] = [5, 6]
```

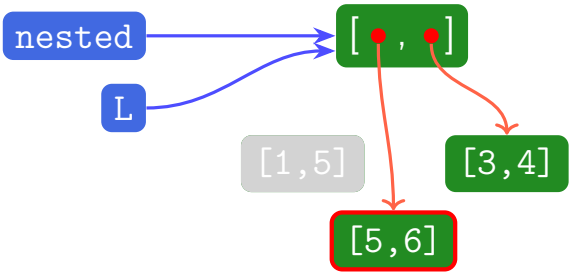
After: `print(nested)` **(the change made through** L **shows!)**



Nested Lists and Aliasing

```
1 nested = [[1, 2], [3, 4]]
2 L = nested
3 L[0][1] = 5
4 print(nested)  # Output:  [[1, 5], [3, 4]]
5 L[0] = [5, 6]  ←
```

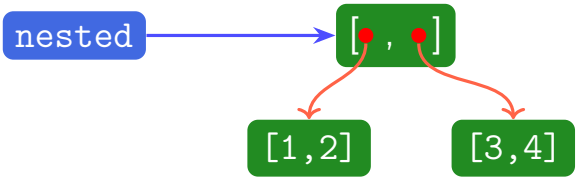
After: `L[0] = [5, 6]` (*L* and *nested*, both change!)



Cloning Nested List (Shallow Copy)

```
1 nested = [[1, 2], [3, 4]] ←
2 L = nested.copy()  # or L = nested[:]
3 L[0][1] = 5
4 L[0] = [5, 6]
5 L[0][1] = 7
```

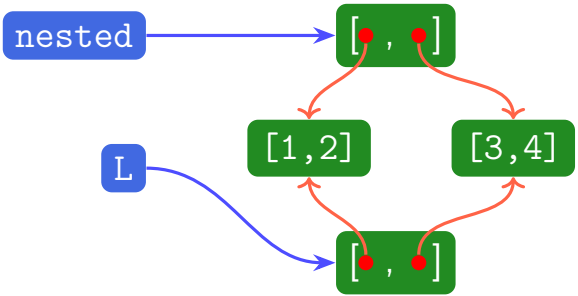
After: `nested = [[1, 2], [3, 4]]`



Cloning Nested List (Shallow Copy)

```
1 nested = [[1, 2], [3, 4]]
2 L = nested.copy()  # or L = nested[:] ←
3 L[0][1] = 5
4 L[0] = [5, 6]
5 L[0][1] = 7
```

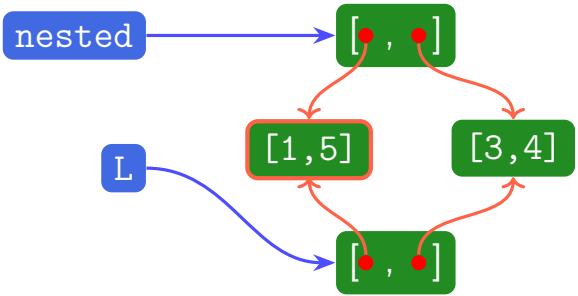
After: `L = nested.copy()` **(new list!)**



Cloning Nested List (Shallow Copy)

```
1 nested = [[1, 2], [3, 4]]
2 L = nested.copy()  # or L = nested[:]
3 L[0][1] = 5 ←
4 L[0] = [5, 6]
5 L[0][1] = 7
```

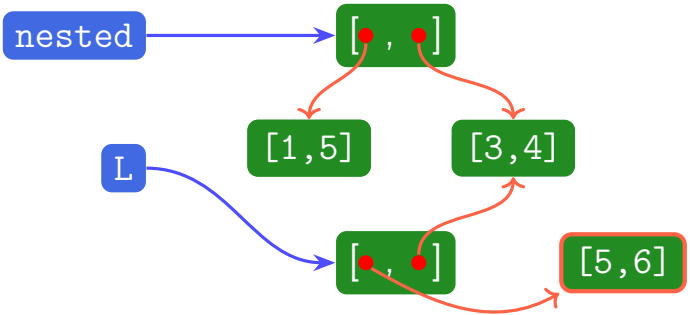
After: `L[0][1] = 5` (*nested* changes too!)



Cloning Nested List (Shallow Copy)

```
1 nested = [[1, 2], [3, 4]]
2 L = nested.copy()  # or L = nested[:]
3 L[0][1] = 5
4 L[0] = [5, 6] ←
5 L[0][1] = 7
```

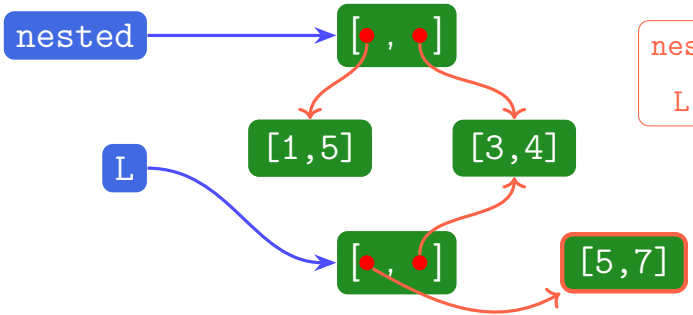
After: `L[0] = [5, 6]` **(only *L* changes!)**



Cloning Nested List (Shallow Copy)

```
1 nested = [[1, 2], [3, 4]]
2 L = nested.copy()  # or L = nested[:]
3 L[0][1] = 5
4 L[0] = [5, 6]
5 L[0][1] = 7 ←
```

After: `L[0][1] = 7` **(only *L* changes!)**



`nested` does **NOT** change:

`L[0]` is L's own list now

You Try!

On paper: what do the two `print` lines show?

```
A = [1, [2, 3], 4]
B = A[:]  # shallow copy
B[0] = 7
B[1][0] = 99
print(A)
print(B)
```

You Try!

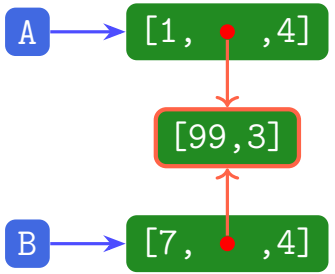
On paper: what do the two `print` lines show?

```
A = [1, [2, 3], 4]
B = A[:]  # shallow copy
B[0] = 7
B[1][0] = 99
print(A) → [1, [99, 3], 4]
print(B) → [7, [99, 3], 4]
```

You Try!

On paper: what do the two `print` lines show?

```
1  A = [1, [2, 3], 4]
2  B = A[:]  # shallow copy
3  B[0] = 7
4  B[1][0] = 99
5  print(A) → [1, [99, 3], 4]
6  print(B) → [7, [99, 3], 4]
```



✓ `B[0] = 7` changes B's **own** outer list, so `A` keeps its 1.

✗ `B[1][0] = 99` changes the inner list. There is only **one** inner list, so `A` shows 99 too.

Deep Copy

- As we saw, **shallow copy** (e.g., using `list.copy()` or slicing) only creates a new outer list, but inner lists are still shared (aliased).

Deep Copy

- As we saw, **shallow copy** (e.g., using `list.copy()` or slicing) only creates a new outer list, but inner lists are still shared (aliased).
- To create a completely independent copy of a nested list (including all inner lists), we need a **deep copy**.

Deep Copy

- As we saw, **shallow copy** (e.g., using `list.copy()` or slicing) only creates a new outer list, but inner lists are still shared (aliased).
- To create a completely independent copy of a nested list (including all inner lists), we need a **deep copy**.
- Python's `copy` module provides a `deepcopy()` function for this.

Deep Copy

- As we saw, **shallow copy** (e.g., using `list.copy()` or slicing) only creates a new outer list, but inner lists are still shared (aliased).
- To create a completely independent copy of a nested list (including all inner lists), we need a **deep copy**.
- Python's `copy` module provides a `deepcopy()` function for this.

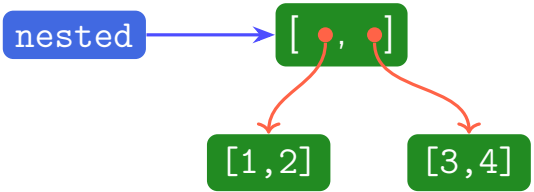
```
import copy
nested = [[1, 2], [3, 4]]
L = copy.deepcopy(nested)  # Deep copy
L[0][1] = 5
print(nested)  # Output: [[1, 2], [3, 4]]
print(L)       # Output: [[1, 5], [3, 4]]
```

Note: `copy.copy(nested)` is the **shallow** copy: the same as `nested[:]`.

Deep Copy (Complete Independence)

```
1 nested = [[1, 2], [3, 4]]
2 L = copy.deepcopy(nested)
3 L[0][1] = 5
```

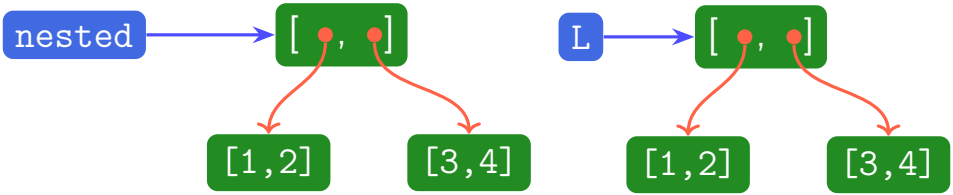
After: `nested = [[1, 2], [3, 4]]`



Deep Copy (Complete Independence)

```
1 nested = [[1, 2], [3, 4]]
2 L = copy.deepcopy(nested)
3 L[0][1] = 5
```

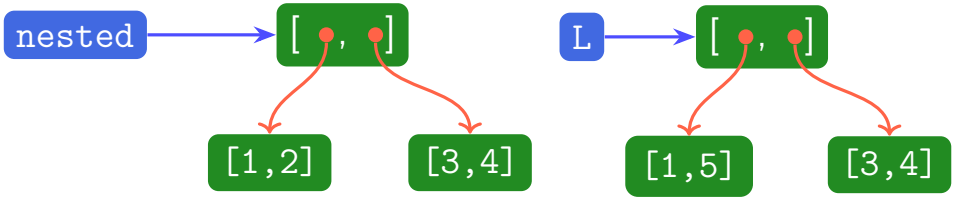
After: `L = copy.deepcopy(nested)` **(completely new!)**



Deep Copy (Complete Independence)

```
1 nested = [[1, 2], [3, 4]]
2 L = copy.deepcopy(nested)
3 L[0][1] = 5
```

nested is **completely independent** from L



Summary of List Methods

Method	Description
`append(x)`	Adds item `x` to the end of the list.
`extend(iterable)`	Extends the list by appending elements from the iterable.
`insert(i, x)`	Inserts item `x` at position `i`.
`remove(x)`	Removes the first occurrence of item `x`.
`pop(i)`	Removes and returns item at position `i` (default last).
`clear()`	Removes all items from the list.

Summary

- Avoid adding/removing items from a list while iterating over it.
- Use slicing (`[:]`) or `list.copy()`
- A list holds **references** (arrows): = and * copy arrows, not objects.
- Understand aliasing and copying (shallow vs deep).
- == asks for the same contents; `is` asks for the same object.
- Lists are mutable and can be modified in functions.
- Nested lists can be complex; use deep copy for full independence.

Questions?

Exit Ticket

Today's question *(2 minutes)*

```
c = [2, 4, 6]
d = c
c[0] = 99
```

(a) What is `d` now?

(b) If line 2 were `d = c[:]`, what would `d` be?

COMP 102 — Exit Ticket #12

Name: _____

Roll #: _____ Date: _____

Answer:

Hand it to me on your way out.
No name, no attendance.