Lecture 05

References and Mutable Data

Luna received a duck. The duck is mutable (note missing limbs)

hw02 technical notes

Testing for expected outcome

- `assert my_function() == b`

  This works for **integers**, **strings**, and **Booleans**

What about floating point numbers?

- assert 3.14159265 - eps < calc_pi() and

  calc_pi() < 3.14159265 + eps

  Note: ‘eps’ should be some small floating point number (e.g., 1e-6)

hw02 technical notes (con’t)

Conversions of data types and constants:

- `float()`
- `int()`
- `str()`

- `float(‘inf’)`
- `-float(‘inf’)`
- `float(‘nan’)`

Checking the Python version:

Add this to the top of your .py script

```
import sys
print(sys.version)
```

This should produce version 3.x.y

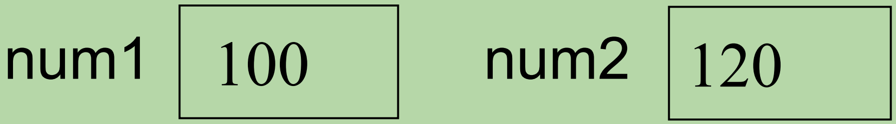
```
3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516]
```

Recall: Variables as Boxes

- You can picture a variable as a named "box" in memory.

- Example from an early lecture:

```
num1 = 100
num2 = 120
```

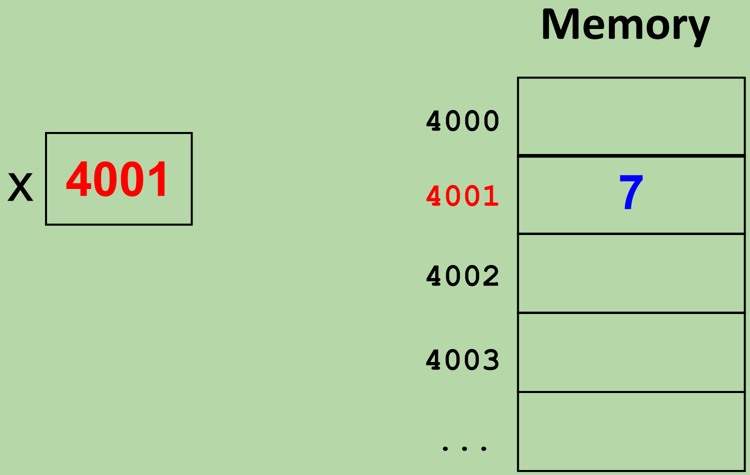


num1 100

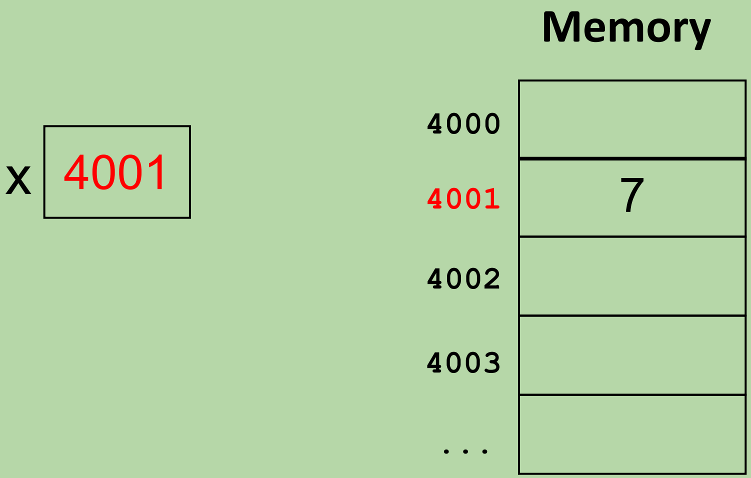
num2 120

Variables and Values

- In Python, when we assign a value to a variable, we're not actually storing the value *in* the variable.
- Rather:
  - the value is somewhere else in memory
  - the variable stores the *memory address* of the value.
- Example: x = 7



References



x 4001

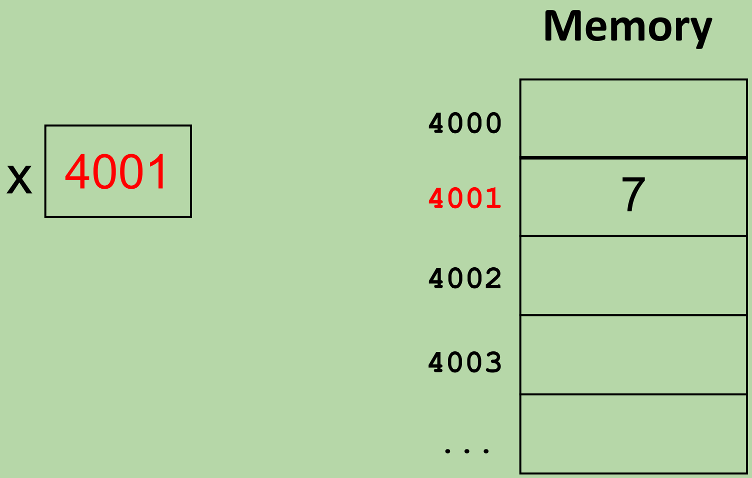
Memory

4000	
4001	7
4002	
4003	
. . .	

We say that a variable stores a *reference* to its value.

- also known as a *pointer*

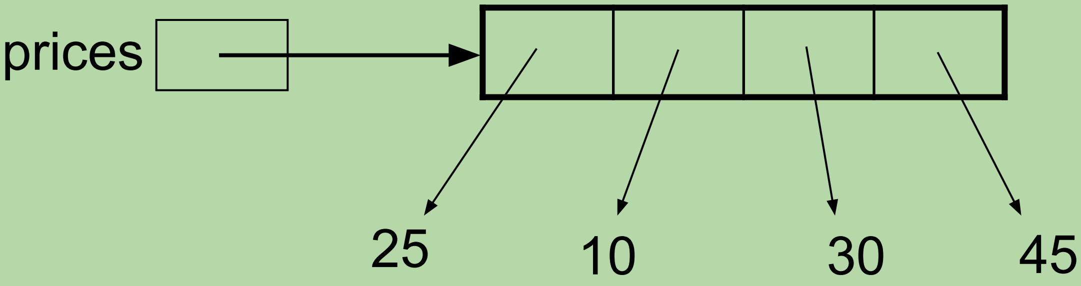
References (cont.)



- Because we don't care about the actual memory address, we use an arrow to represent a reference:

Lists and References

```
prices = [25, 10, 30, 45]
```



- When a variable represents a list, it stores a reference to the list.

- The list itself is a *collection* of references!
  - each element of the list is a reference to a value

Mutable vs. Immutable Data

- In Python, strings and numbers are *immutable*.
  - their contents/components cannot be changed

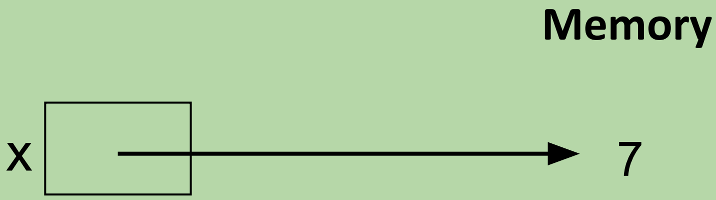
- Lists are *mutable.*
  - their contents/components *can* be changed
  - example:

```
>>> prices = [25, 10, 30, 45]
>>> prices[2] = 50
>>> print(prices)
[25, 10, 50, 45]
```

Changing a Value vs. Changing a Variable

- There's no way to change an immutable value like 7.

x = 7



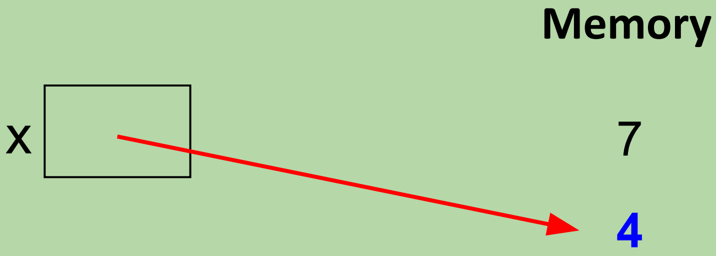
Memory

x

7

- However, we *can* use assignment to change the variable—making it refer to a different value:

x = 4



Memory

x

7

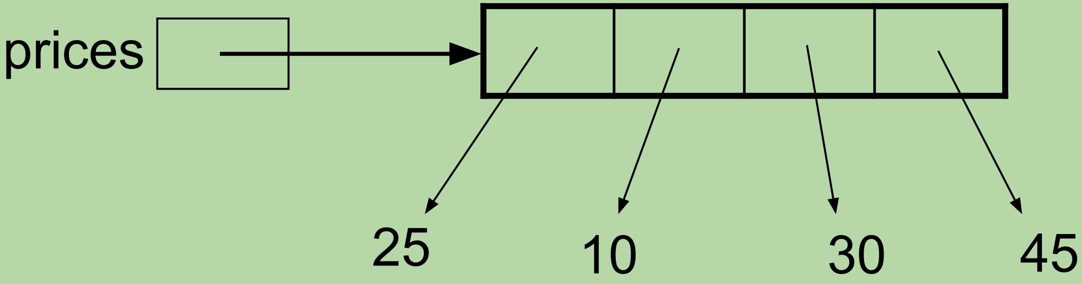
4

- We're not actually changing the value 7.

- We're making the variable x refer to a different value.

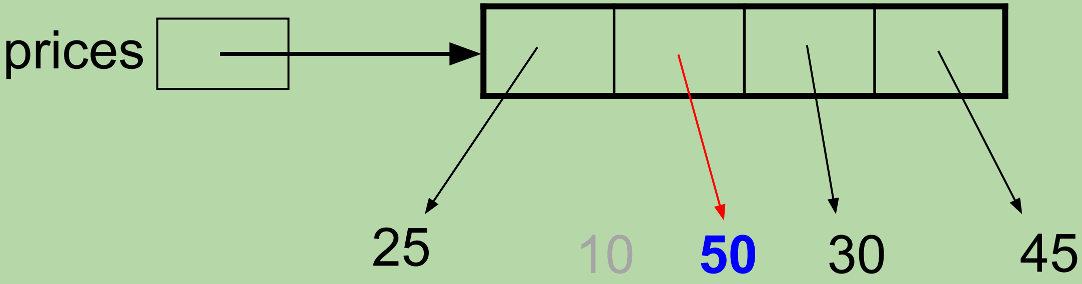
Changing a Value vs. Changing a Variable

- Here's our original list:



- Lists are mutable, so we *can* change the value (the list) by modifying its elements:

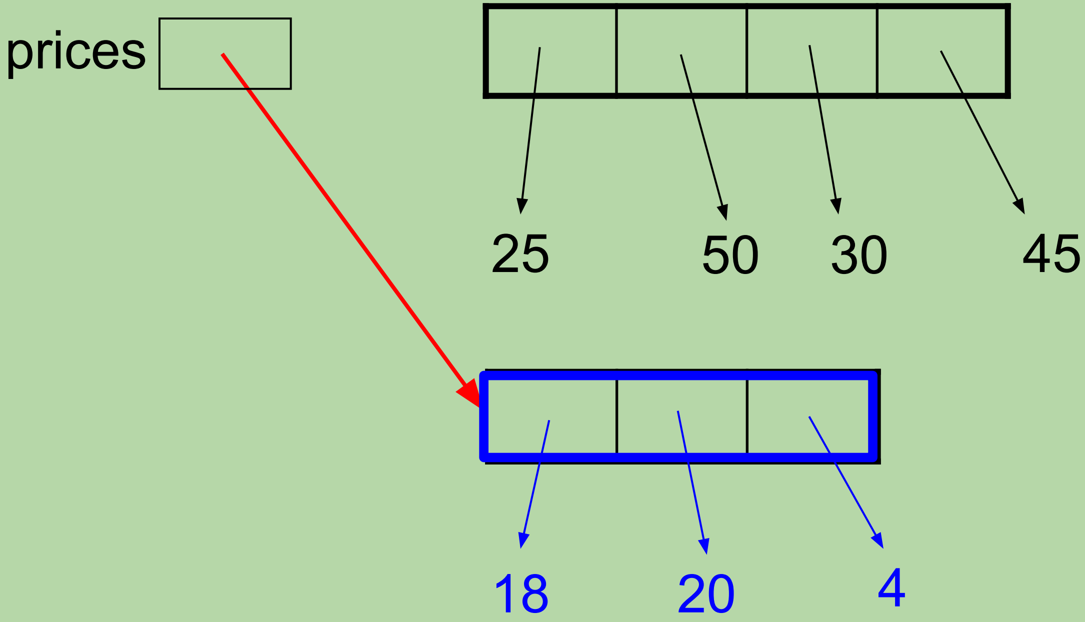
`prices[1] = 50`



Changing a Value vs. Changing a Variable

- We can also change the variable—making it refer to a completely different list:

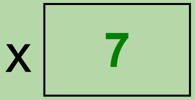
prices = [18, 20, 4]



Simplifying Our Mental Model

- When a variable represents an immutable value,
  it's okay to picture the value as being *inside* the variable.

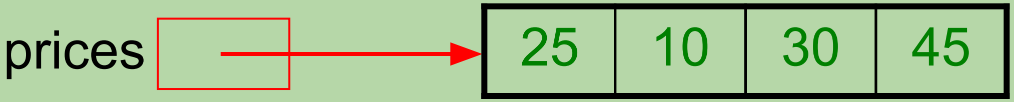
```
x = 7
```



  - a simplified picture, but good enough!

- The same thing holds for list elements that are immutable.

```
prices = [25, 10, 30, 45]
```



- We still need to use references for *mutable* data like lists.

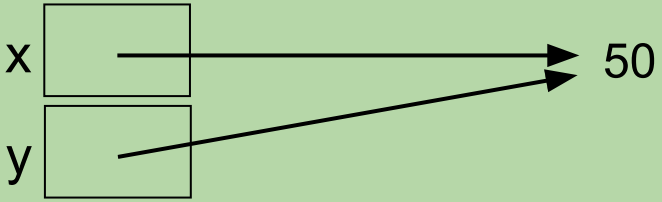
Copying Variables

- The assignment

```
var2 = var1
```

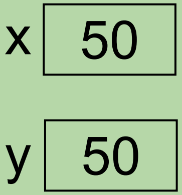
copies the reference of *var1* into *var2, e.g.,*

```
x = 50
    y = x
```



- But when the data is in var1 is immutable you can use the box notation, e.g.,

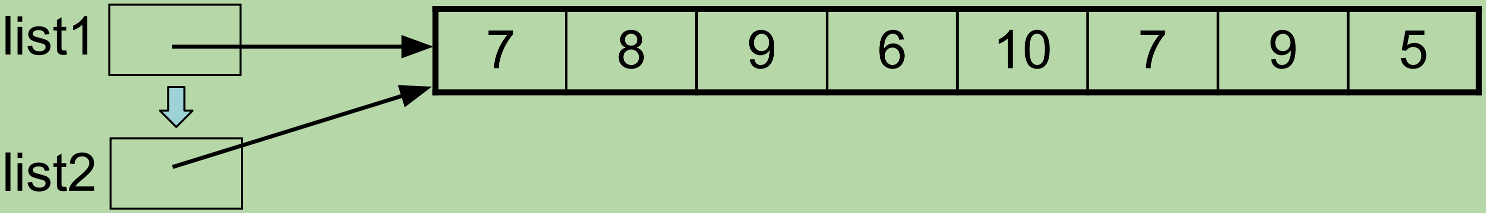
```
x = 50
    y = x
```



Copying References

- Consider this example:

```
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1
```



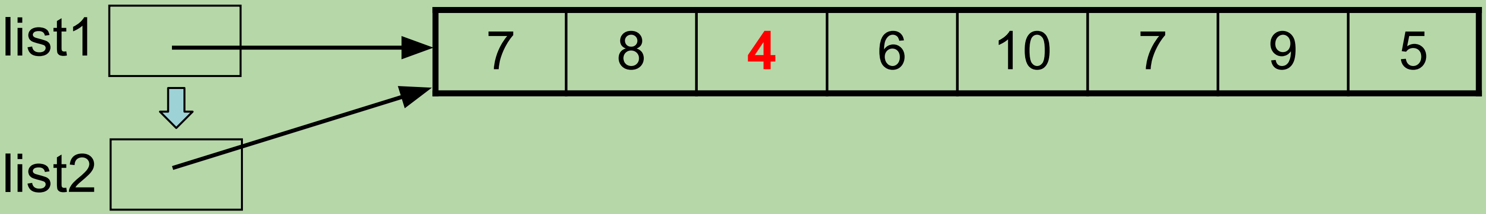
- Given the lines of code above, what will the lines below print?

```
list2[2] = 4
print(list1[2], list2[2])
```

Copying References

- Consider this example:

```
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1
```



- Given the lines of code above, what will the lines below print?

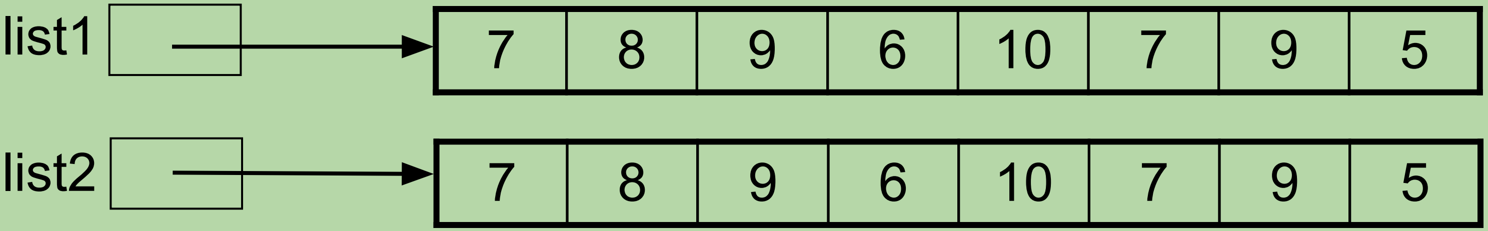
```
list2[2] = 4
print(list1[2], list2[2])
                 4       4
                 4 4
```

- Copying a list variable simply copies the reference.
- It doesn't copy the list itself!

Copying a List, using slicing

- We can copy a list like slicing:

```
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1[:]
```



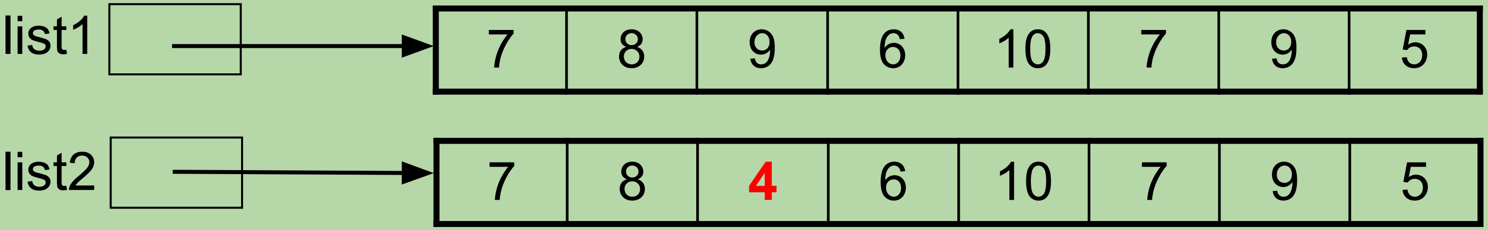
- What will this print now?

```
 list2[2] = 4
print(list1[2], list2[2])
```

Copying a List, using slicing

- We can copy a list like this one using a full slice:

```
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1[:]
```



- What will this print now?

```
list2[2] = 4
  print(list1[2], list2[2])
             9    4
                9 4
```

What does this program output?

```
list1 = [1, 2, 3]
list2 = list1[:]
list3 = list2
list2[1] = 7
print(list1, list2, list3)
```

A. [1, 2, 3] [1, 7, 3] [1, 2, 3]

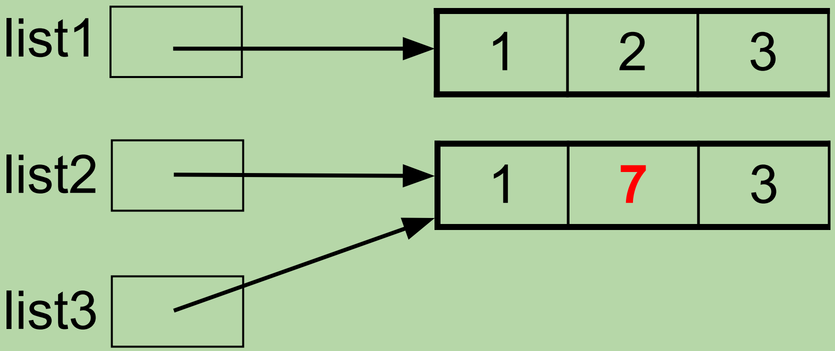
B. [1, 7, 3] [1, 7, 3] [1, 2, 3]

C. [1, 2, 3] [1, 7, 3] [1, 7, 3]

D. [1, 7, 3] [1, 7, 3] [1, 7, 3]

What does this program output?

```
list1 = [1, 2, 3]
list2 = list1[:]
list3 = list2
list2[1] = 7
print(list1, list2, list3)
```



A. [1, 2, 3] [1, 7, 3] [1, 2, 3]

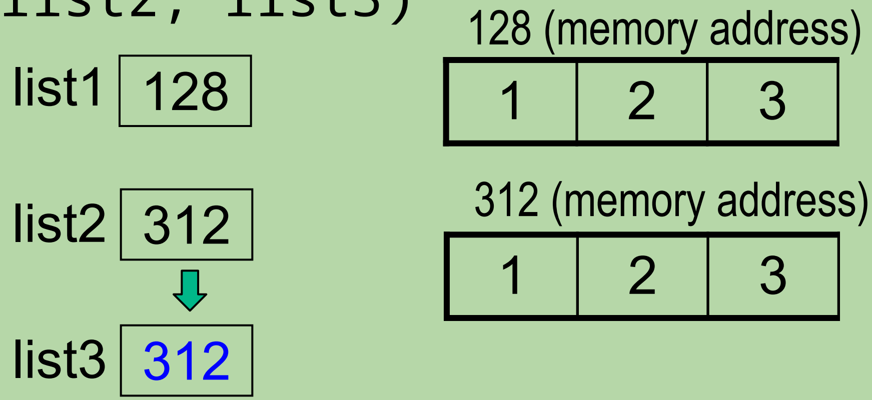
B. [1, 7, 3] [1, 7, 3] [1, 2, 3]

C. **[1, 2, 3] [1, 7, 3] [1, 7, 3]**

D. [1, 7, 3] [1, 7, 3] [1, 7, 3]

What does this program output?

```
list1 = [1, 2, 3]
list2 = list1[:]
list3 = list2
list2[1] = 7
print(list1, list2, list3)
```



list1 128

list2 312

list3 312

128 (memory address)

1	2	3

312 (memory address)

1	2	3

A. [1, 2, 3] [1, 7, 3] [1, 2, 3]

B. [1, 7, 3] [1, 7, 3] [1, 2, 3]

C. **[1, 2, 3] [1, 7, 3] [1, 7, 3]**

D. [1, 7, 3] [1, 7, 3] [1, 7, 3]

Passing an Immutable Value to a Function

- When an immutable value (like a number or string) is passed into a function, we can think of the function as getting a copy of value (though really it gets a reference).

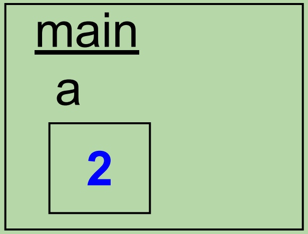
- If the function changes its copy of the value, that change will *not* be there when the function returns, this is because any assignment to the local variable updates it's reference and not the referenced value.

- Consider the following program:

```
def main():
      a = 2
      triple(a)
      print(a)    # what will be printed?


  def triple(x):
      x = x * 3
```

Passing an Immutable Value to a Function (cont.)

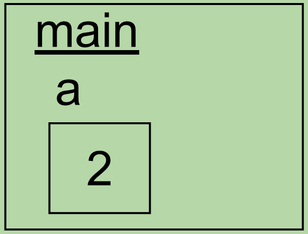
before call to triple()



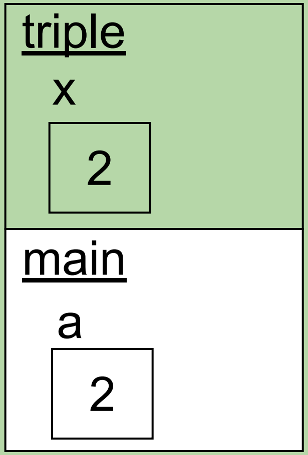
```
def triple(x):
        x = x * 3

def main():
    a = 2
    triple(a)
    print(a)
```

Passing an Immutable Value to a Function (cont.)

before call to triple()



during call to triple()

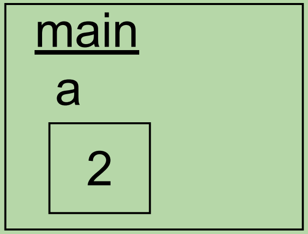


```
def triple(x):
        x = x * 3

def main():
    a = 2
    triple(a)
    print(a)
```

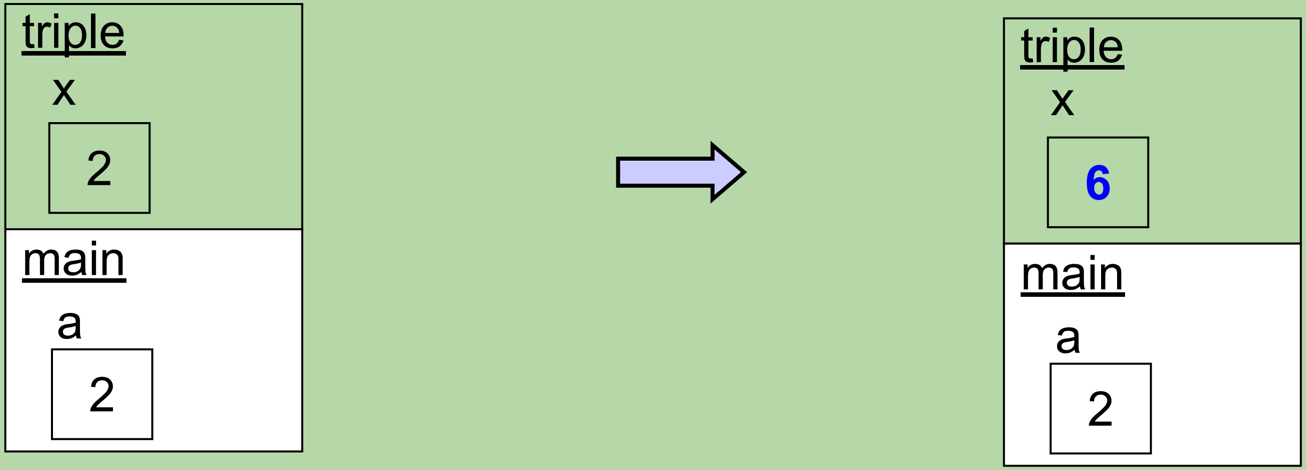
Passing an Immutable Value to a Function (cont.)

```
def triple(x):
    x = x * 3

def main():
  a = 2
  triple(a)
  print(a)
```

before call to triple()



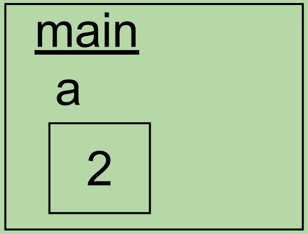
during call to triple()



Passing an Immutable Value to a Function (cont.)

```
def main():
    a = 2
    triple(a)
    print(a)       # prints 2
```

before call to triple()

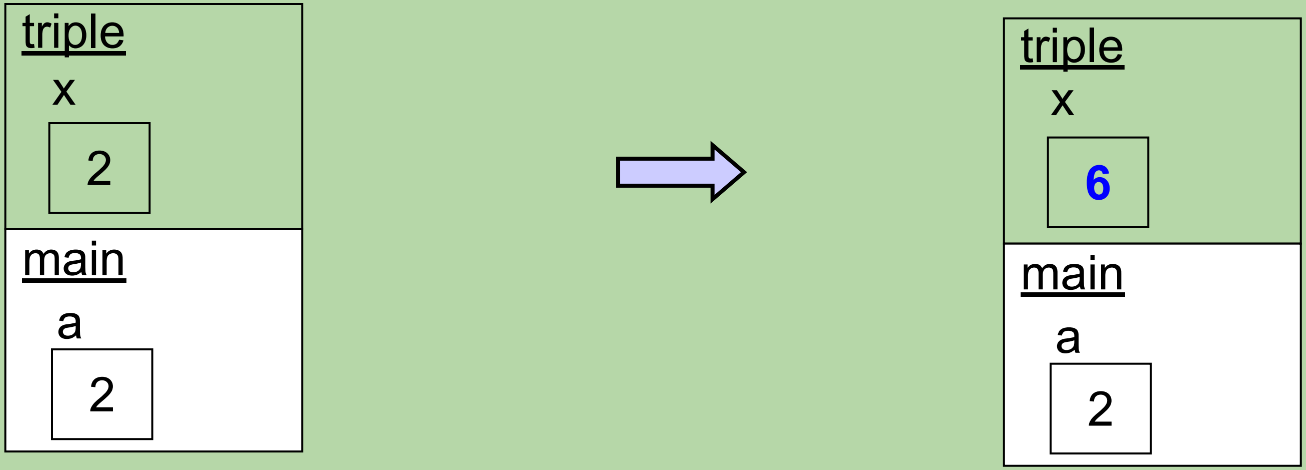


main

a

2

during call to triple()



triple

x

2

main

a

2

triple

x

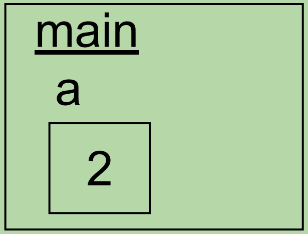
6

main

a

2

after call to triple()



main

a

2

Passing a List to a Function

- When a list is passed into a function:
  - the function gets a copy of the *reference* to the list
  - it does *not* get a copy of the list itself

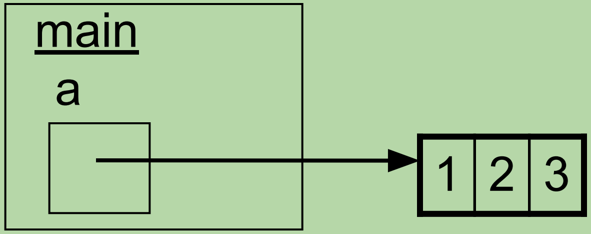
- Thus, if the function changes the components of the list, those changes will be there when the function returns.

- Consider the following program:

```
def main():
    a = [1, 2, 3]
    triple(a)
    print(a)    # what will be printed?

def triple(vals):
    for i in range(len(vals)):
        vals[i] = vals[i] * 3
```

Passing a List to a Function (cont.)

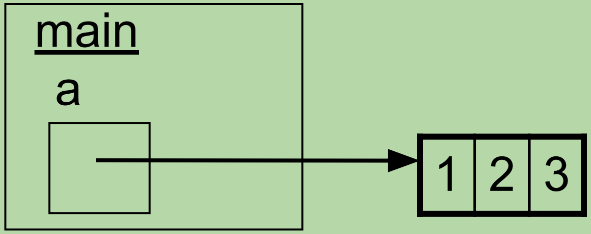
before call to triple()



```
def main():
    a = [1, 2, 3]
    triple(a)
    print(a)
```

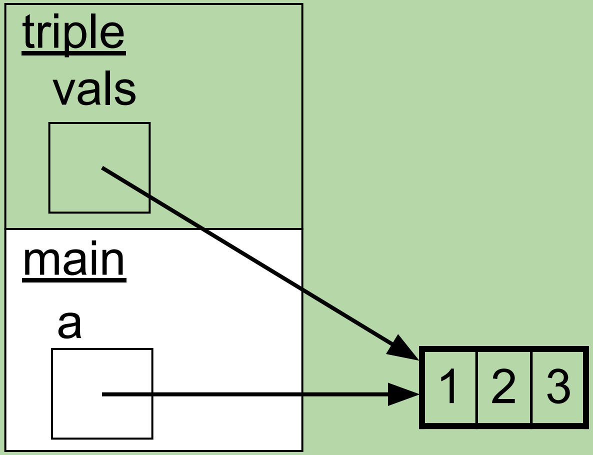
Passing a List to a Function (cont.)

before call to triple()



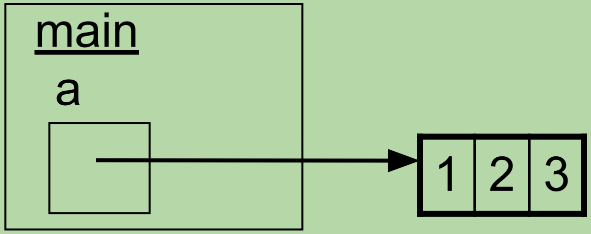
```
def main():
    a = [1, 2, 3]
    triple(a)
    print(a)
```

during call to triple()



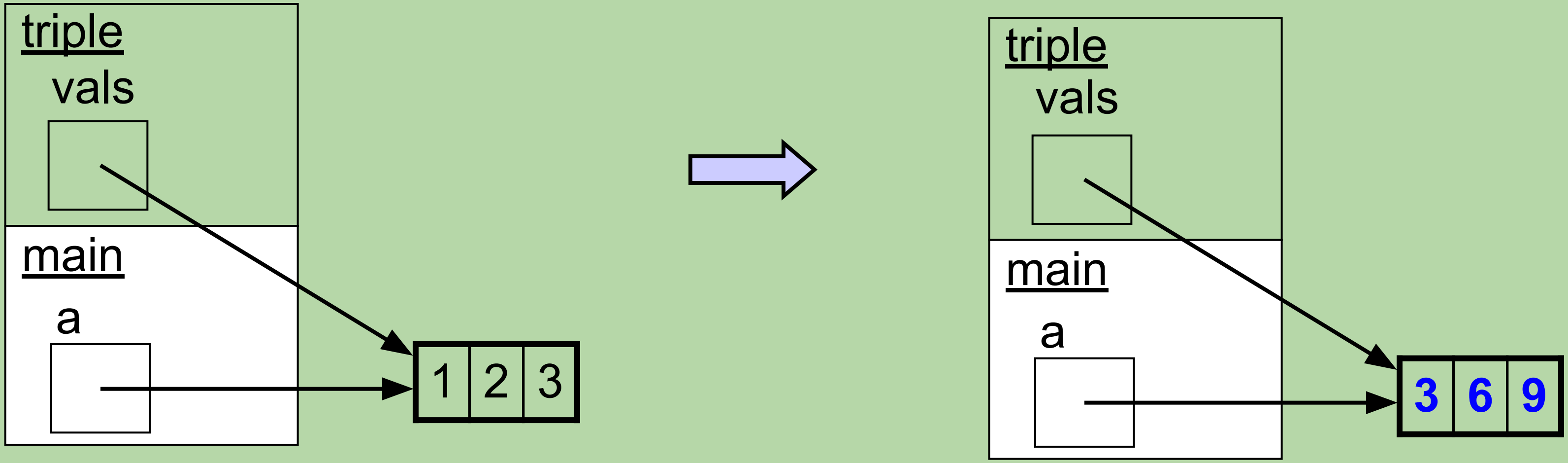
Passing a List to a Function (cont.)

before call to triple()



```
def triple(vals) :
    for i in range(len(vals)):
        vals[i] = vals[i] * 3
```

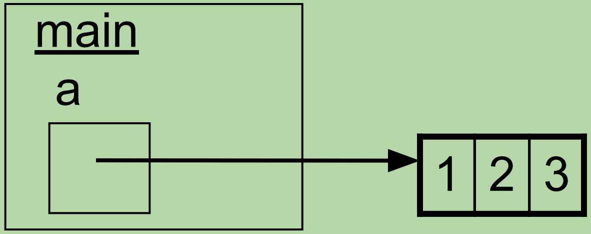
during call to triple()



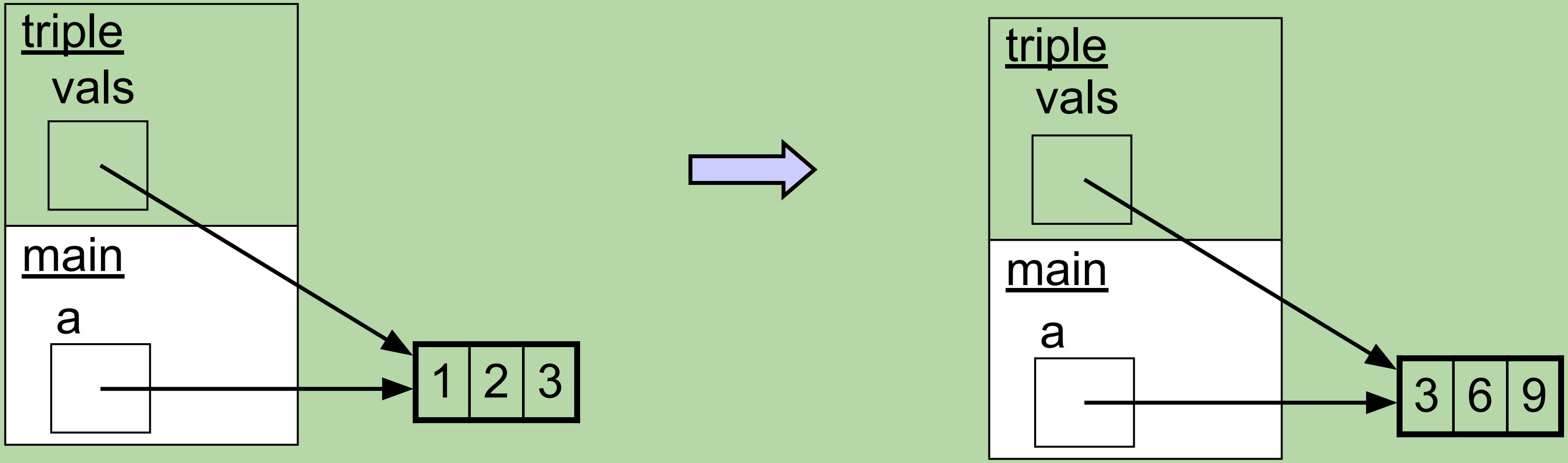
Passing a List to a Function (cont.)

```
def main():
    a = [1, 2, 3]
    triple(a)
    print(a)       # prints [3, 6, 9]
```

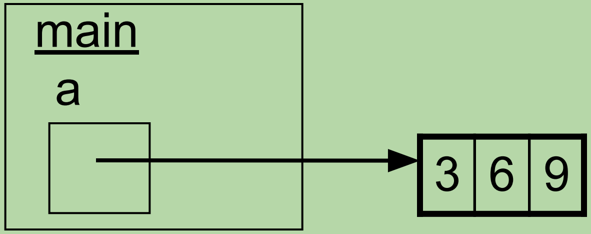
before call to triple()



during call to triple()



after call to triple()



What does this program output?

```
def mystery1(x):
    x *= 2
    return x
def mystery2(vals):
    vals[0] = 111
    return vals

x = 7
vals = [7, 7]
mystery1(x)
mystery2(vals)
print(x, vals)
```

A. 7 [7, 7]

B. 14 [7, 7]

C. 7 [111, 7]

D. 14 [111, 7]

What does this program output?

```
def mystery1(x):
    x *= 2
    return x
def mystery2(vals):
    vals[0] = 111
    return vals

x = 7
vals = [7, 7]
mystery1(x)
mystery2(vals)
print(x, vals)
```

A. 7 [7, 7]

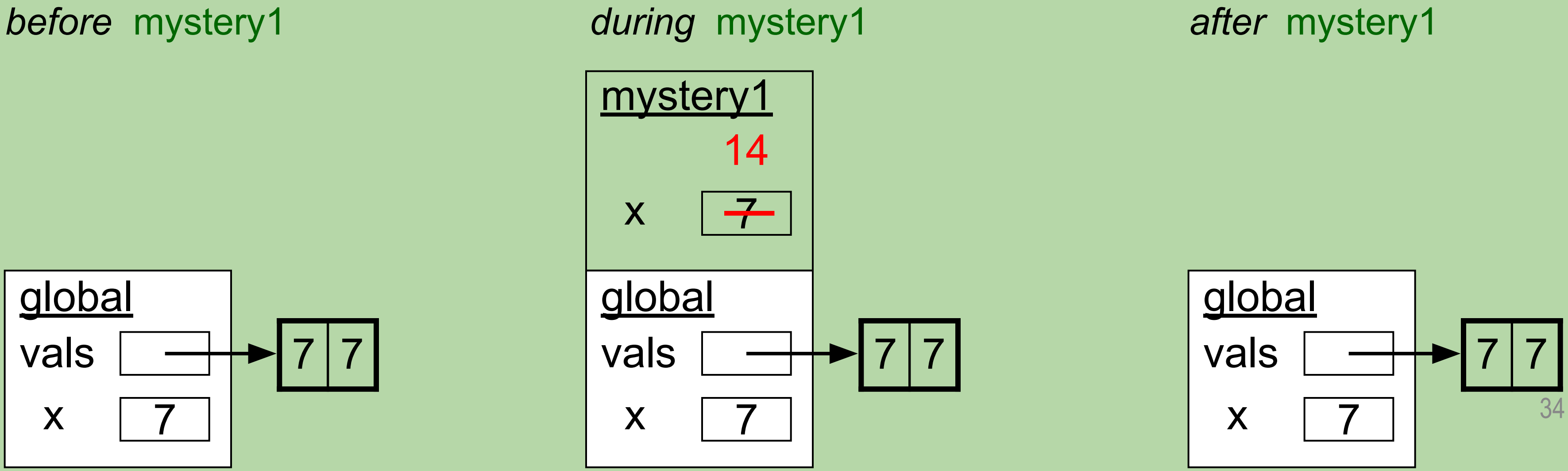
B. 14 [7, 7]

C. **7 [111, 7]**

D. 14 [111, 7]

What does this program output?

```
def mystery1(x):
    x *= 2
    return x
def mystery2(vals):
    vals[0] = 111
    return vals

x = 7
vals = [7, 7]
mystery1(x)     # throws return value away!
mystery2(vals)
print(x, vals)
```

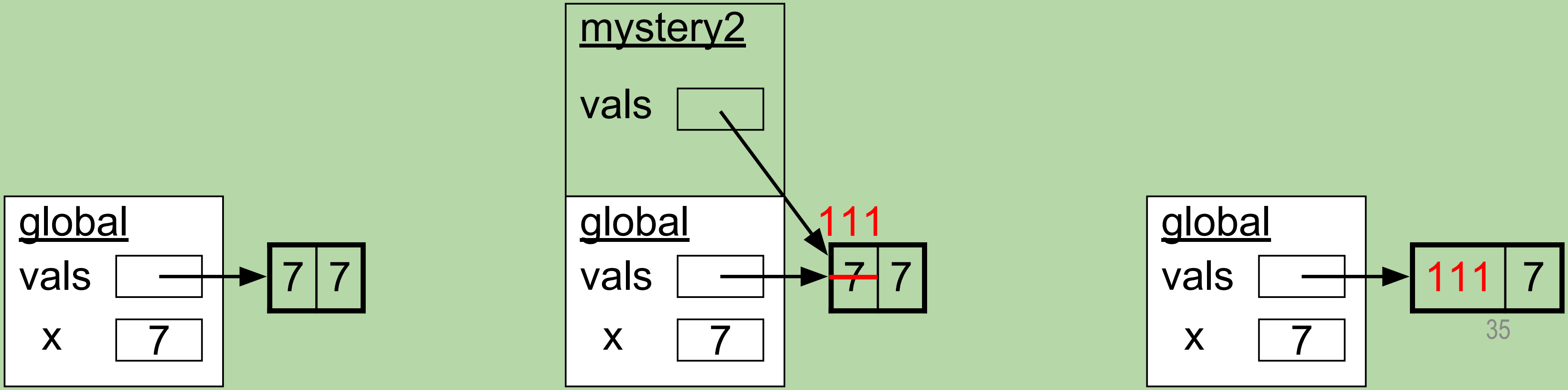


What does this program output?

```
def mystery1(x):
    x *= 2
    return x
def mystery2(vals):
    vals[0] = 111
    return vals

x = 7
vals = [7, 7]
mystery1(x)
mystery2(vals)
print(x, vals)        # output: 7 [111, 7]
```

---

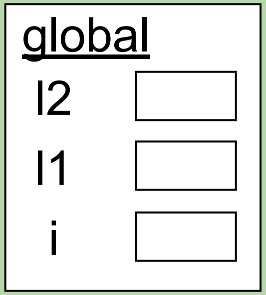
before mystery2 *during* mystery2 *after* mystery2



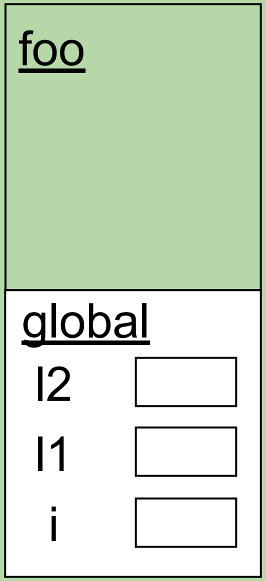
What does this program print?
Draw your own memory diagrams!

```
def foo(vals, i):
    i += 1
    vals[i] *= 2

i = 0
l1 = [1, 1, 1]
l2 = l1
foo(l2, i)
print(i, l1, l2)
```

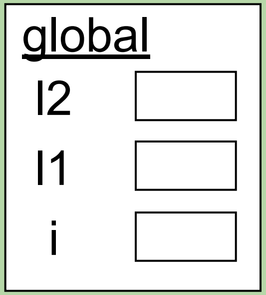
before foo



during foo



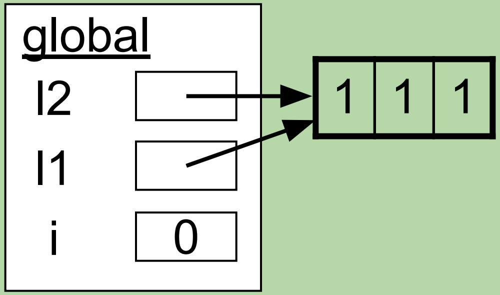
after foo



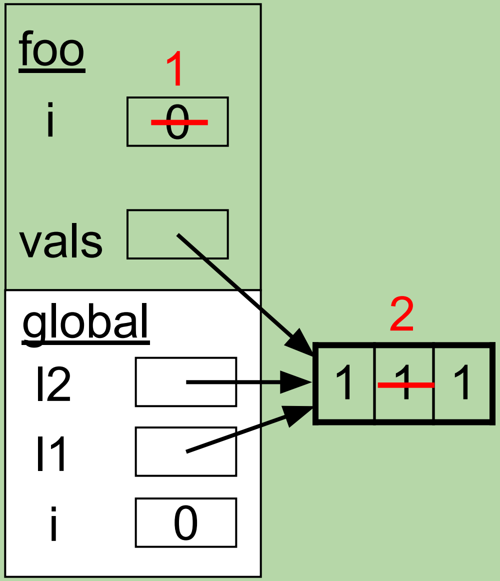
What does this program print?
Draw your own memory diagrams!

```
def foo(vals, i):
    i += 1
    vals[i] *= 2

i = 0
l1 = [1, 1, 1]
l2 = l1
foo(l2, i)
print(i, l1, l2)      # output: 0 [1, 2, 1] [1, 2, 1]
```

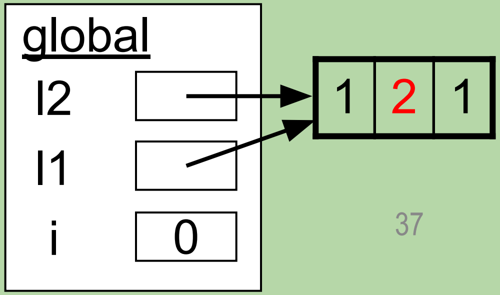
before foo



during foo



after foo



Extra practice: What about this program?

```
def mystery3(x):
    x = 111
    return x
def mystery4(vals):
    vals = [111, 111]
    return vals

x = 7
vals = [7, 7]
mystery3(x)
mystery4(vals)
print(x, vals)
```

A. 7 [7, 7]

B. 111 [7, 7]

C. 7 [111, 111]

D. 111 [111, 111]

Extra practice: What about this program?

```
def mystery3(x):
    x = 111
    return x
def mystery4(vals):
    vals = [111, 111]
    return vals

x = 7
vals = [7, 7]
mystery3(x)
mystery4(vals)
print(x, vals)
```

A. **7 [7, 7]**

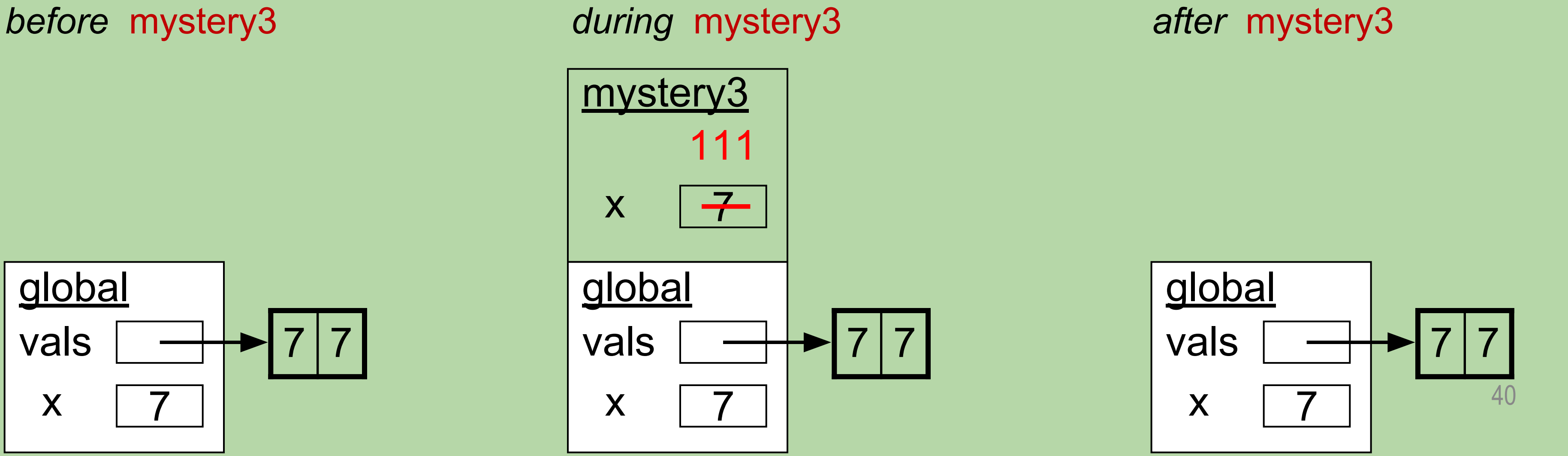
B. 111 [7, 7]

C. 7 [111, 111]

D. 111 [111, 111]

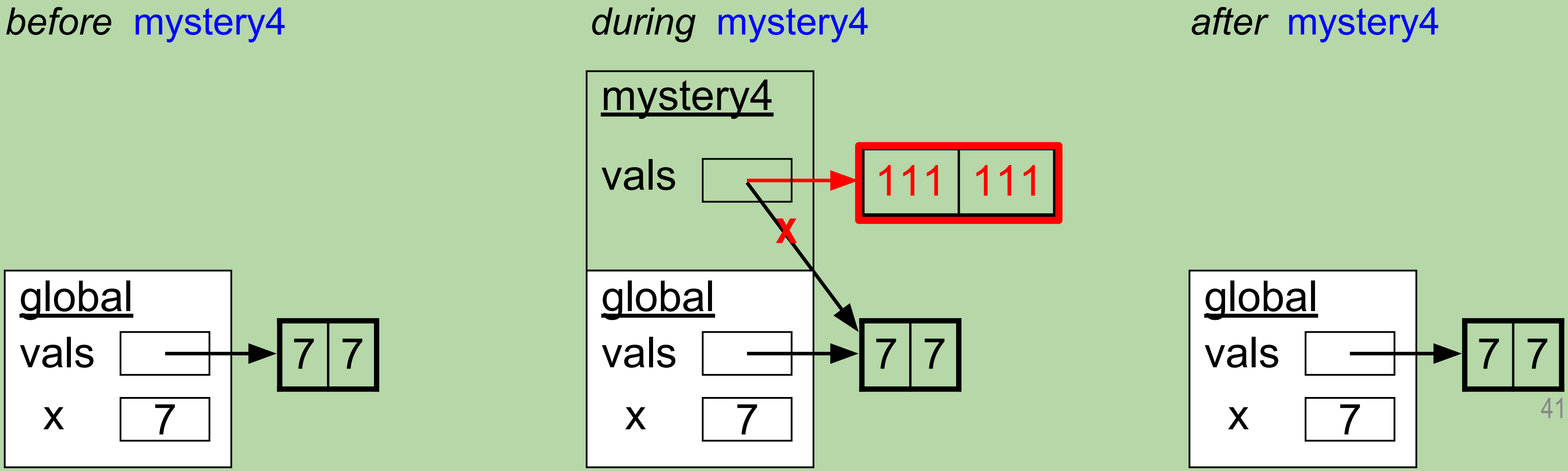
Extra practice: What about this program?

```
def mystery3(x):
    x = 111
    return x
def mystery4(vals):
    vals = [111, 111]
    return vals

x = 7
vals = [7, 7]
mystery3(x)     # throw return value away!
mystery4(vals)
print(x, vals)
```



Extra practice: What about this program?

```
def mystery3(x):
    x = 111
    return x
def mystery4(vals):
    vals = [111, 111]
    return vals

x = 7
vals = [7, 7]
mystery3(x)
mystery4(vals)
print(x, vals)     # output: 7 [7, 7]
```



Recall Our Earlier Example...

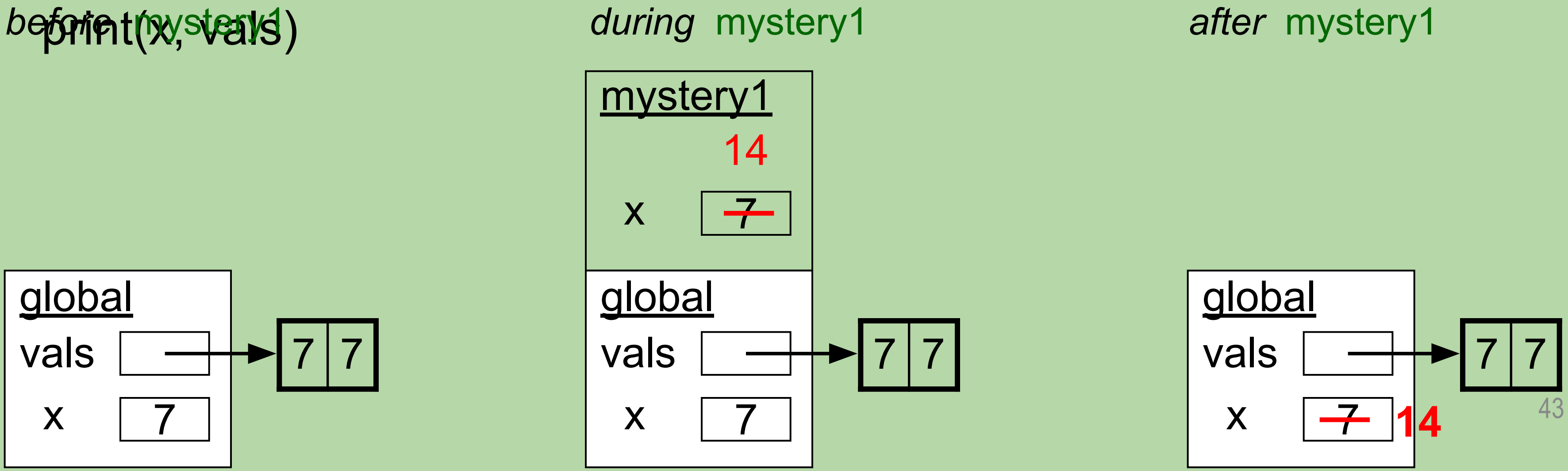
```
def mystery1(x):
    x *= 2
    return x
def mystery2(vals):
    vals[0] = 111
    return vals

x = 7
vals = [7, 7]
mystery1(x)
mystery2(vals)
print(x, vals)
```

How can we make the *global* `x` reflect `mystery1`'s change?

Recall Our Earlier Example...

```
def mystery1(x):
    x *= 2
    return x
def mystery2(vals):
    vals[0] = 111
    return vals

x = 7
vals = [7, 7]
x = mystery1(x)    # assign the return value!
mystery2(vals)
print(x, vals)
```

How can we make the *global* x reflect mystery1's change?


before mystery1 | *during* mystery1 | *after* mystery1

2-D Lists

based in part on notes from the CS-for-All curriculum
developed at Harvey Mudd College

2-D Lists

- Recall that a list can include sublists

```
mylist = [17, 2, [2, 5], [1, 3, 7]]
```

What is `len(mylist)`?

2-D Lists

- Recall that a list can include sublists

```
mylist = [17, 2, [2, 5], [1, 3, 7]]
```

What is len(mylist)? 4

- To capture a rectangular table or grid of values, use a *two-dimensional* list:

```
table = [[15,  8,  3, 16, 12,  7,  9   5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

  - a list of sublists, each with the same length
  - each sublist is one "row" of the table

2-D Lists: Try These Questions!

```
table = [[15,  8,  3, 16, 12,  7,  9   5],
     [ 6, 11,  9,  4,  1,  5,  8, 13],
     [17,  3,  5, 18, 10,  6,  7, 21],
     [ 8, 14, 13,  6, 13, 12,  8,  4],
     [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

- what is len(table)?
- what does table[0] represent?

  table[1]?

  table[-1]?
- what is len(table[0])?
- what is table[3][1]?

- how would you change the 1 in the lower-left corner to a 7?

2-D Lists: Try These Questions!

```
table = [[15,  8,  3, 16, 12,  7,  9   5],
     [ 6, 11,  9,  4,  1,  5,  8, 13],
     [17,  3,  5, 18, 10,  6,  7, 21],
     [ 8, 14, 13,  6, 13, 12,  8,  4],
     [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

- what is len(table)? 5 (more generally, the # of rows / height)
- what does table[0] represent? the first/top row

  table[1]? the second row

  table[-1]? the last/bottom row
- what is len(table[0])? 8 (the # of columns / width)
- what is table[3][1]? 14

  row index ↗ ↖ column index
- how would you change the 1 in the lower-left corner to a 7?

  `table[4][0] = 7      # table[-1][0] = 7 also works!`

Dimensions of a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9   5],
      [ 6, 11,  9,  4,  1,  5,  8, 13],
      [17,  3,  5, 18, 10,  6,  7, 21],
      [ 8, 14, 13,  6, 13, 12,  8,  4],
      [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

len(table) is the # of rows in table

table[r] is the row with index r

len(table[r]) is the # of elements in row r

len(table[0]) is the # of columns in table

Picturing a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9   5],
     [ 6, 11,  9,  4,  1,  5,  8, 13],
     [17,  3,  5, 18, 10,  6,  7, 21],
     [ 8, 14, 13,  6, 13, 12,  8,  4],
     [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

- Here's one way to picture the above list:

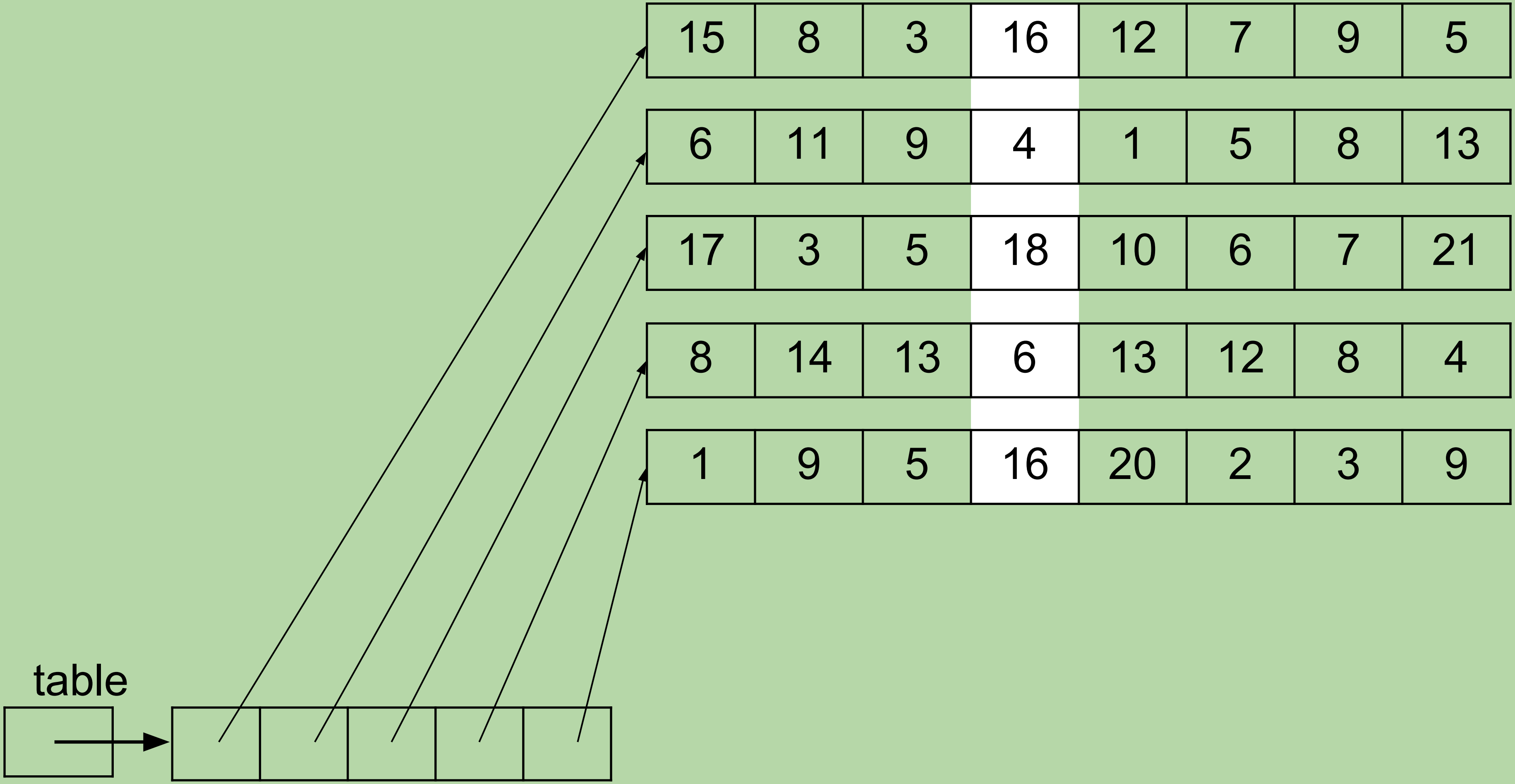
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	15	8	3	16	12	7	9	5
1	6	11	9	4	1	5	8	13
2	17	3	5	18	10	6	7	21
3	8	14	13	6	13	12	8	4
4	1	9	5	16	20	2	3	9

column indices

row indices

Picturing a 2-D List (cont)

- Here's a more accurate picture:



15	8	3	16	12	7	9	5
6	11	9	4	1	5	8	13
17	3	5	18	10	6	7	21
8	14	13	6	13	12	8	4
1	9	5	16	20	2	3	9

Accessing an Element of a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9   5],
     [ 6, 11,  9,  4,  1,  5,  8, 13],
     [17,  3,  5, 18, 10,  6,  7, 21],
     [ 8, 14, 13,  6, 13, 12,  8,  4],
     [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

table[r][c] is the element at row r, column c in table

examples:

```
>>> print(table[2][1])
3
```

column index

row index

Accessing an Element of a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9   5],
     [ 6, 11,  9,  4,  1,  5,  8, 13],
     [17,  3,  5, 18, 10,  6,  7, 21],
     [ 8, 14, 13,  6, 13, 12,  8,  4],
     [ 1,  9,  5, 16, 20,  2,  0,  9]]
```

`table[r][c]` is the element at row r, column c in table

examples:

```
>>> print(table[2][1])
3
```

column index

row index

```
>>> table[-1][-2] = 0
```

Using Nested Loops to Process a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9   5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

```
for r in range(len(table)):
    for c in range(len(table[0])):
        # process table[r][c]
```

Using Nested Loops to Process a 2-D List

```
table = [[15, 19,  3, 16],
     [ 6, 21,  9,  4],
     [17,  3,  5, 18]]
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] > 15:
            count += 1
print(count)
```

---

r	c	table[r][c]	count

Using Nested Loops to Process a 2-D List

```
table = [[15, 19,  3, 16],
     [ 6, 21,  9,  4],
     [17,  3,  5, 18]]
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] > 15:
            count += 1
print(count)                               # prints 5
```

---

r	c	table[r][c]	count
			0
0	0	15	0
0	1	19	1
0	2	3	1
0	3	16	2
1	0	6	2
1	1	21	3
...			
2	0	17	4
...			
2	3	18	5

Which Of These Counts the Number of Evens?

```
table = [[15, 19,  3, 16],
         [ 6, 21,  9,  4],
         [17,  3,  5, 18]]
```

A.

```
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] % 2 == 0:
            count += 1
```

B.

```
count = 0
for r in len(table):
    for c in len(table[0]):
        if c % 2 == 0:
            count += 1
```

C.

```
count = 0
for r in range(len(table[0])):
    for c in range(len(table)):
        if table[r][c] % 2 == 0:
            count += 1
```

D. either A or B E. either A or C

Which Of These Counts the Number of Evens?

```
table = [[15, 19,  3, 16],
         [ 6, 21,  9,  4],
         [17,  3,  5, 18]]
```

A.

```
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] % 2 == 0:
            count += 1
```

B.

```
count = 0
for r in len(table):
    for c in len(table[0]):
        if c % 2 == 0:
            count += 1
```

C.

```
count = 0
for r in range(len(table[0])):
    for c in range(len(table)):
        if table[r][c] % 2 == 0:
            count += 1
```

D. either A or B   E. either A or C

Using Nested Loops to Process a 2-D List

```
table = [[15, 19,  3, 16],
    [ 6, 21,  9,  4],
    [17,  3,  5, 18]]
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] % 2 == 0:
            count += 1
print(count)
```

---

r	c	table[r][c]	count

Using Nested Loops to Process a 2-D List

```
table = [[15, 19,  3, 16],
         [ 6, 21,  9,  4],
         [17,  3,  5, 18]]
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] % 2 == 0:
            count += 1
print(count)                            # prints 4
```

---

r	c	table[r][c]	count
			0
0	0	15	0
0	1	19	0
0	2	3	0
0	3	16	1
1	0	6	2
1	1	21	2
...			
1	3	4	3
...			
2	3	18	4

What is the output of this program?

```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)
mystery6(vals, vals)
print(x, vals)
```

A. 7 [7, 7]

B. -7 [1, 1]

C. 7 [0, 7]

D. 7 [1, 1]

E. -7 [0, 7]

What is the output of this program?

```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)
mystery6(vals, vals)
print(x, vals)
```

A. 7 [7, 7]

B. -7 [1, 1]

C. **7 [0, 7]**

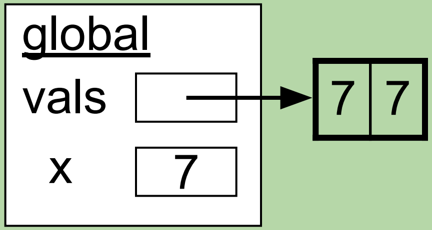
D. 7 [1, 1]

E. -7 [0, 7]

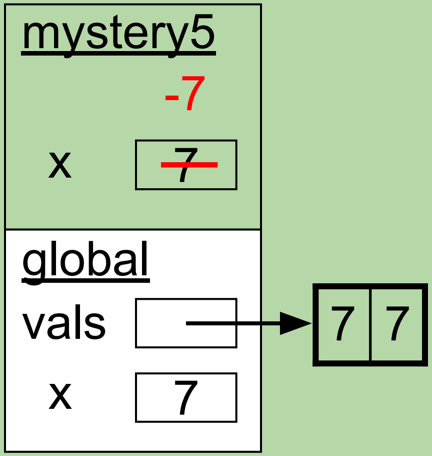
What is the output of this program?

```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)    # throw return value away!
mystery6(vals, vals)
print(x, vals)
```

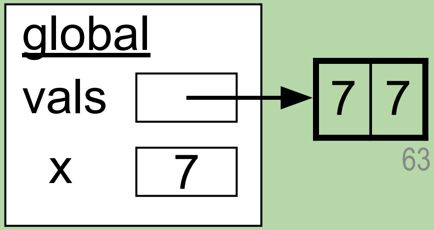
before mystery5



during mystery5



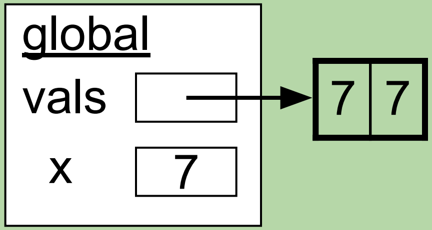
after mystery5



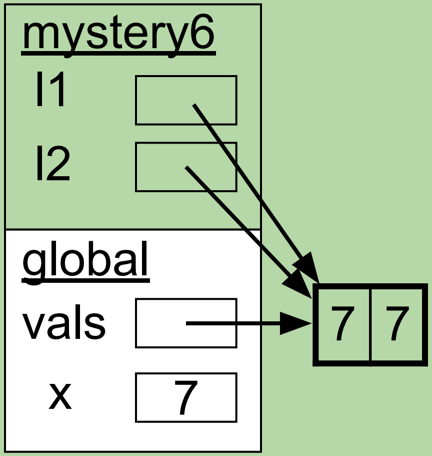
What is the output of this program?

```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)
mystery6(vals, vals)
print(x, vals)
```

before mystery6



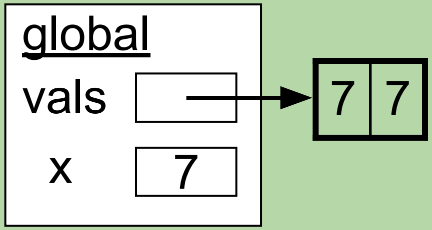
during mystery6



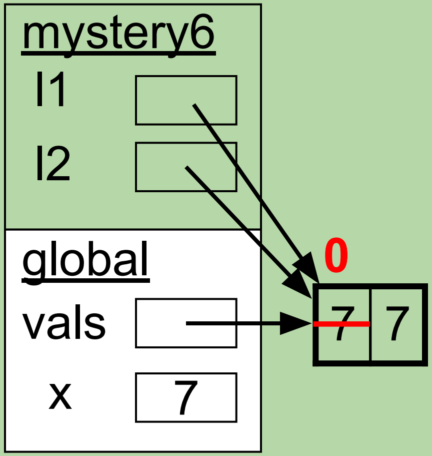
What is the output of this program?

```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)
mystery6(vals, vals)
print(x, vals)
```

before mystery6



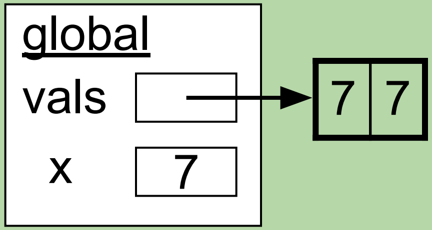
during mystery6



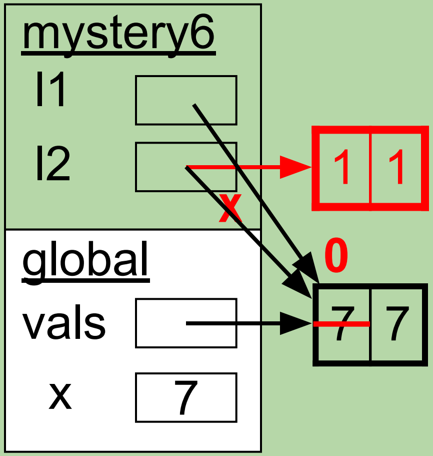
What is the output of this program?

```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)
mystery6(vals, vals)
print(x, vals)
```

before mystery6

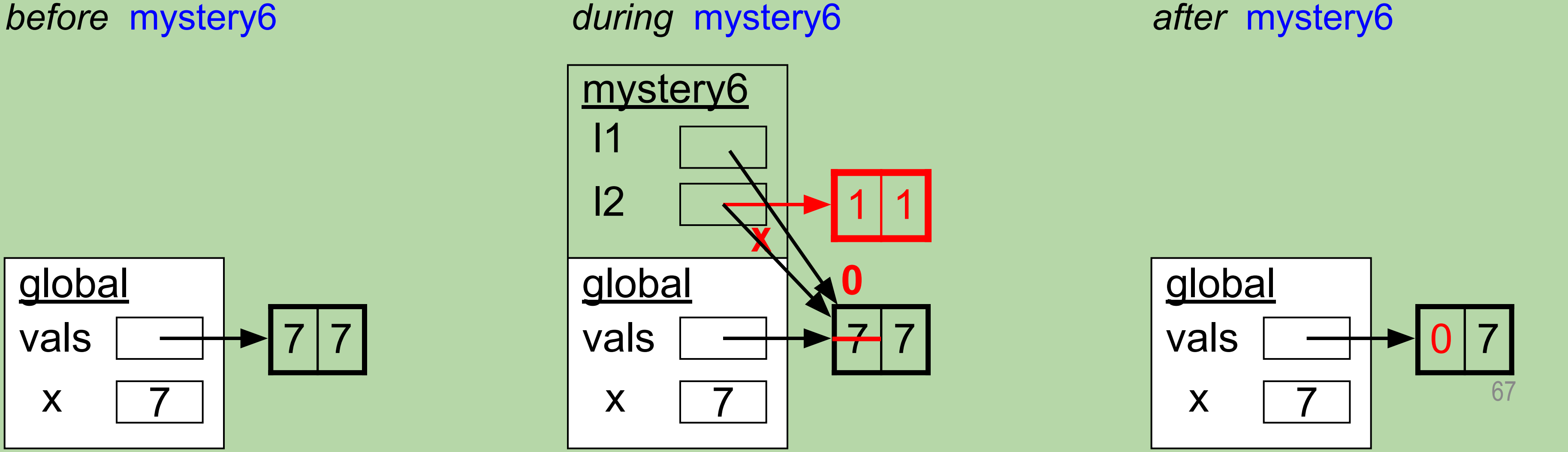


during mystery6



What is the output of this program?

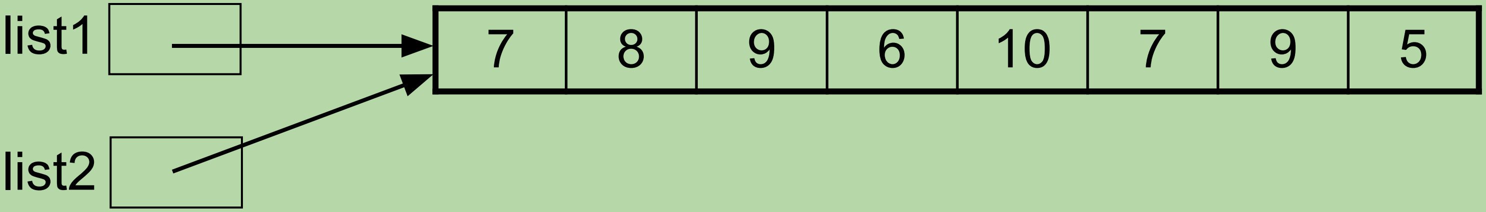
```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)
mystery6(vals, vals)
print(x, vals)       # output: 7 [0, 7]
```



Recall: Copying a List

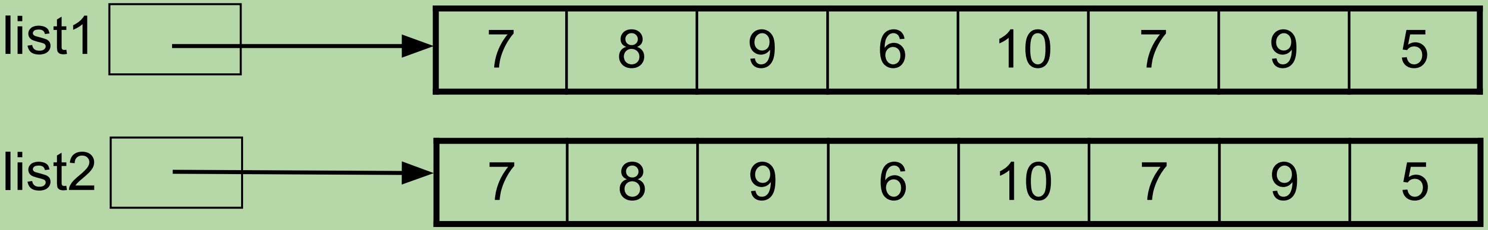
- We can't copy a list by a simple assignment:

```
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1
```



- We can copy this list using a full slice:

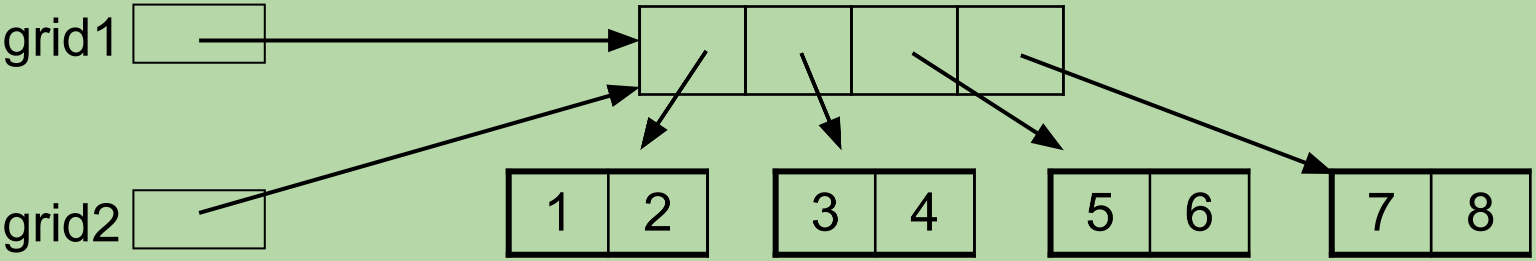
```
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1[:]
```



Copying a 2-D List

`grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]`

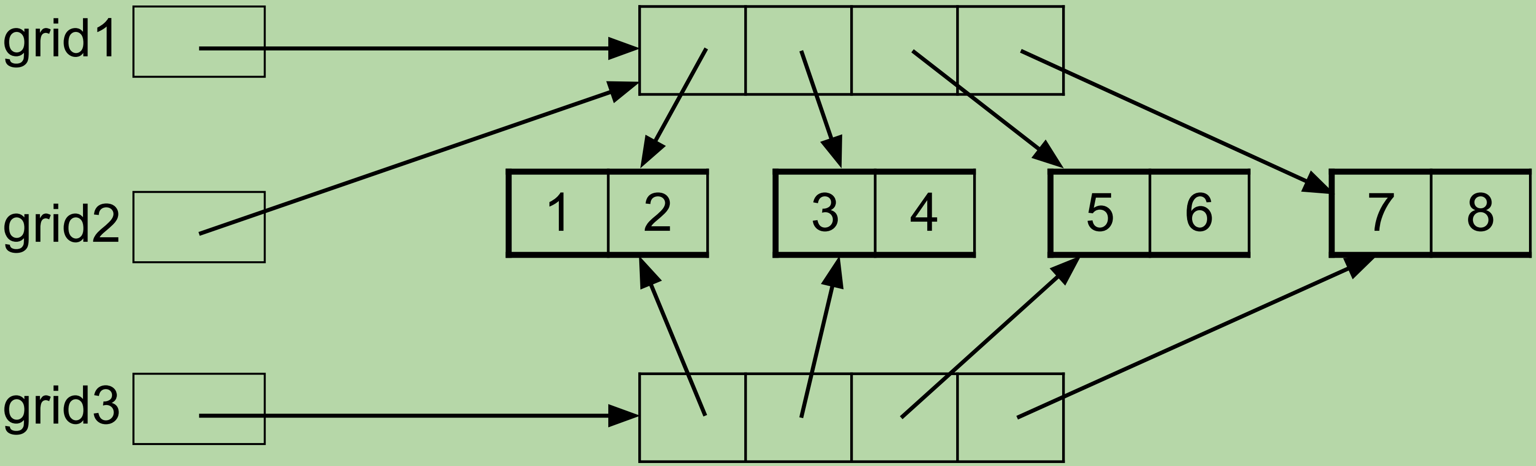
This doesn't copy a list: grid2 = grid1



- Does this? `grid3 = grid1[:]`

Copying a 2-D List

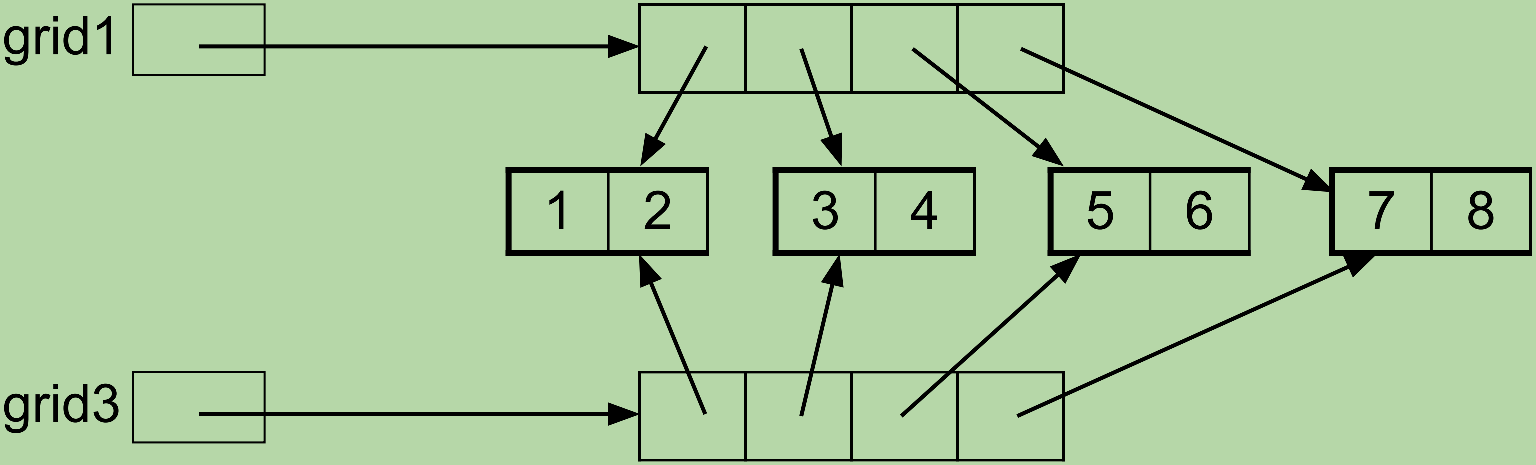
`grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]`



- Does this? `grid3 = grid1`**[:]** **not fully!**

A *Shallow* Copy

```
grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]
grid3 = grid1[:]
```



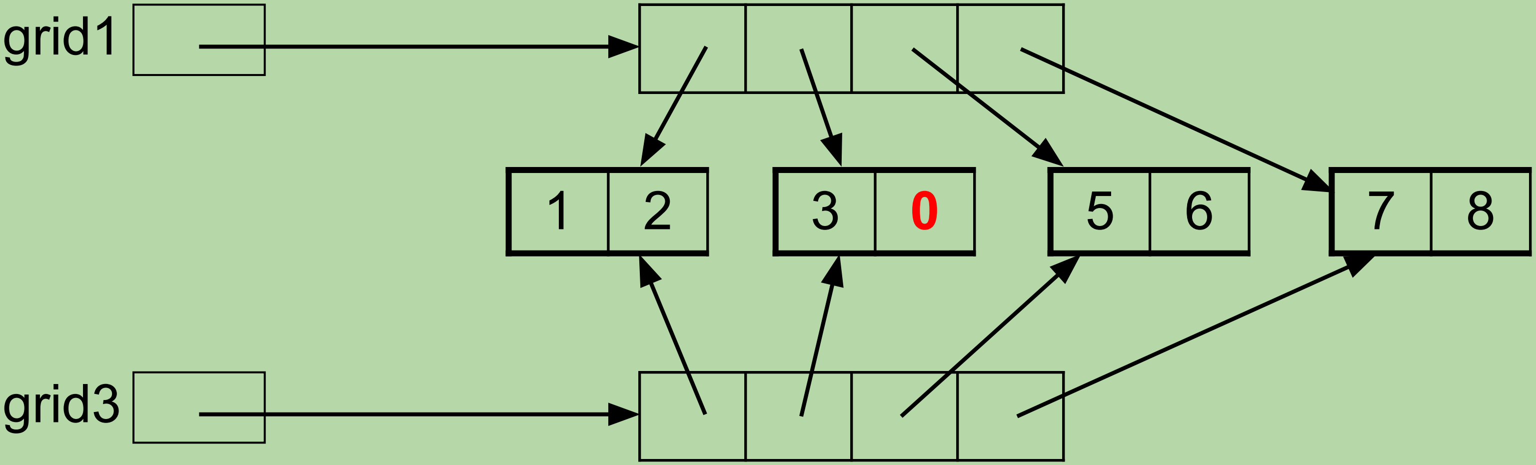
- `grid1` and `grid3` now share the same sublists.
  - known as a *shallow* copy

- What would this print?

```
grid1[1][1] = 0
print(grid3)
```

A *Shallow* Copy

```
grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]
grid3 = grid1[:]
```



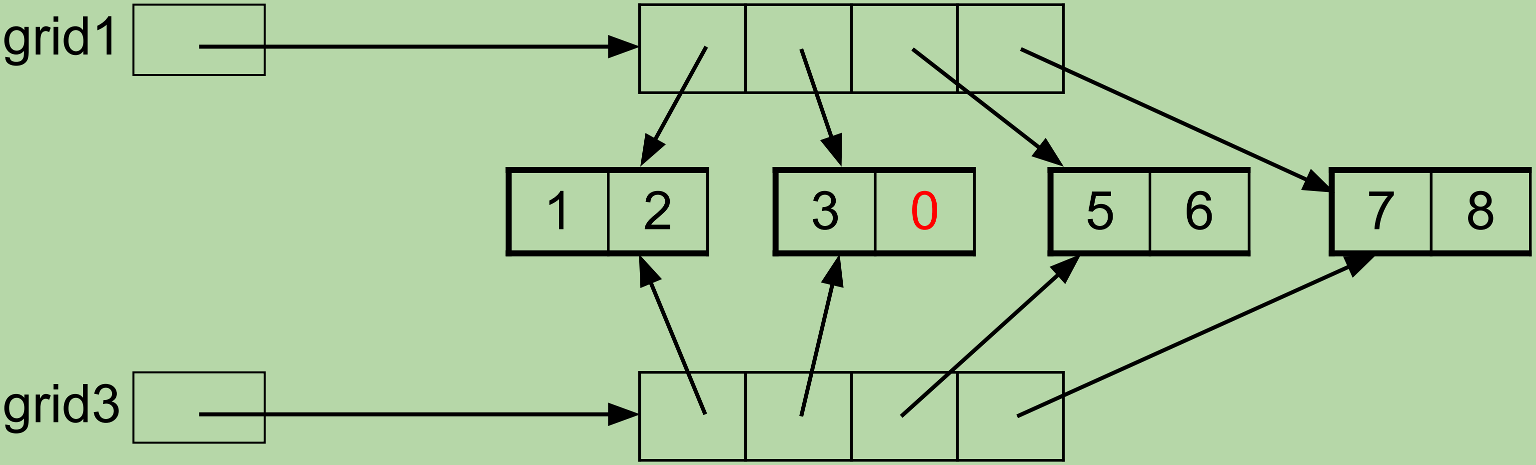
- `grid1` and `grid3` now share the same sublists.
  - known as a *shallow* copy

- What would this print?

```
grid1[1][1] = 0
print(grid3)
```

A *Shallow* Copy

```
grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]
grid3 = grid1[:]
```



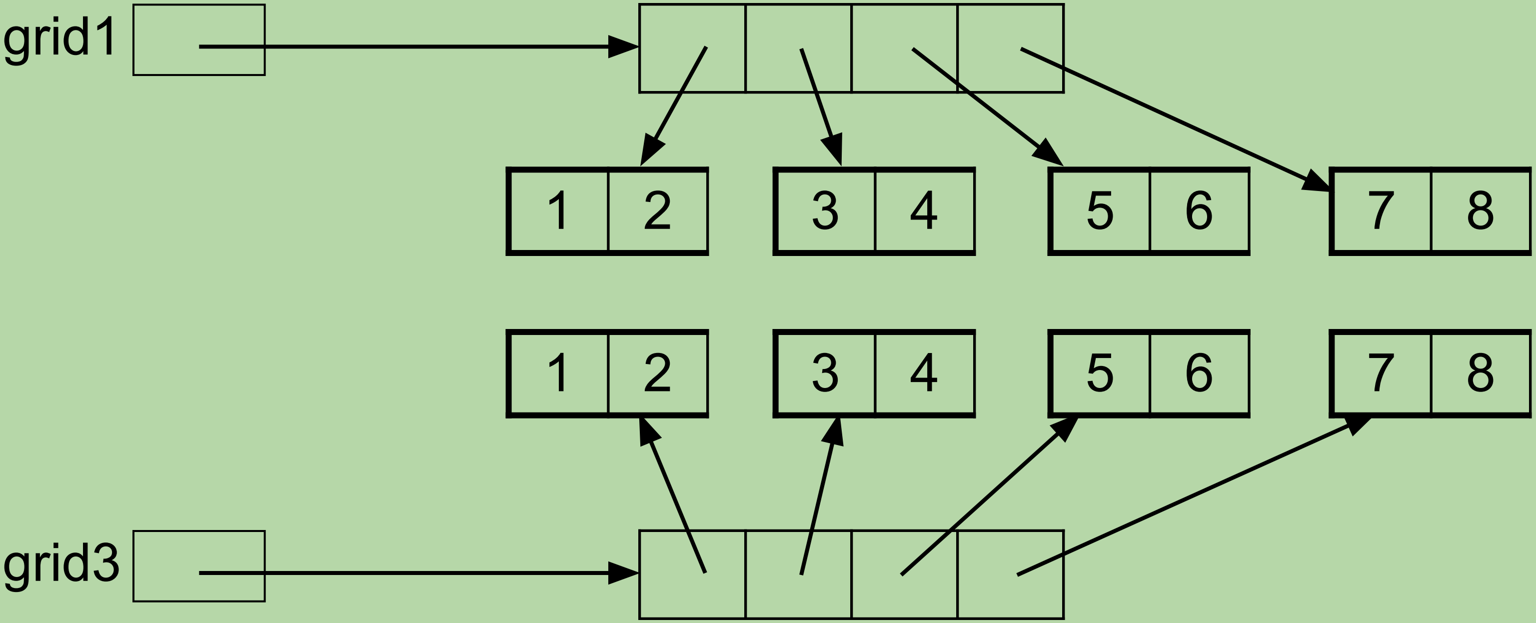
- `grid1` and `grid3` now share the same sublists.
  - known as a *shallow* copy

- What would this print?

```
grid1[1][1] = 0
print(grid3)    [[1, 2], [3, 0], [5, 6], [7, 8]]
```

A *Deep* Copy: Nothing is Shared

```
grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]
```



- Here's one way to achieve this:

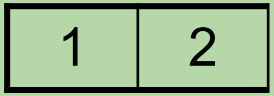
```
grid3 = []
for sublist in grid1:
    grid3 = grid3 + [sublist[:]]
```

In hw03, you'll take a different approach!

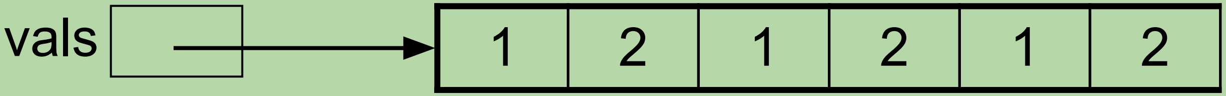
Recall: List Multiplication

```
>>> vals = [1, 2] * 3
>>> vals
[1, 2, 1, 2, 1, 2]
```

- original list:



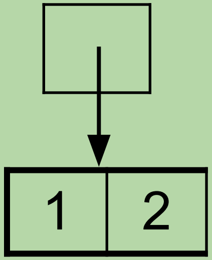
- get 3 copies of it, concatenated together:



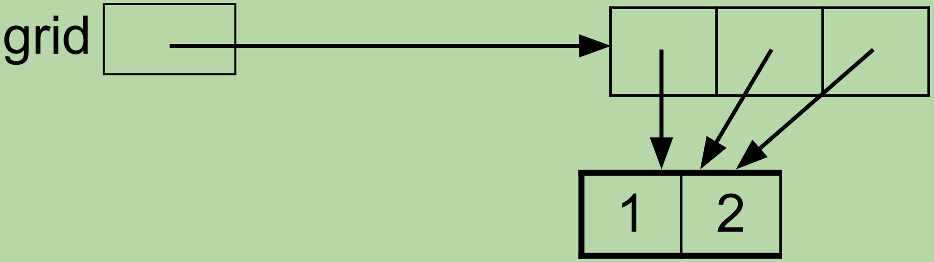
List Multiplication of a 2-D List

```
>>> grid = [[1, 2]] * 3
>>> grid
[[1, 2], [1, 2], [1, 2]]
```

- original list:



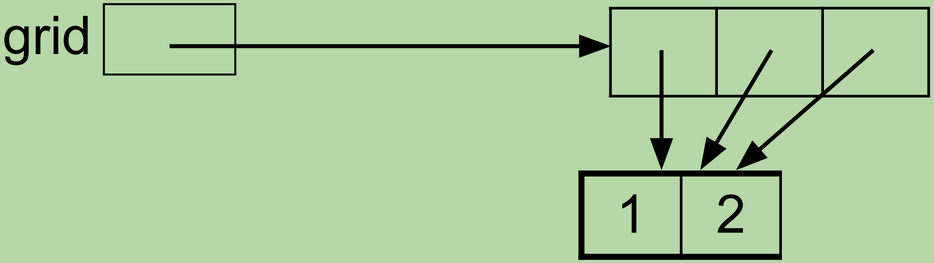
- get 3 copies of it concatenated together:



- the reference to the sublist is copied, not the sublist

List Multiplication of a 2-D List (cont.)

```
>>> grid = [[1, 2]] * 3
>>> grid
[[1, 2], [1, 2], [1, 2]]
```

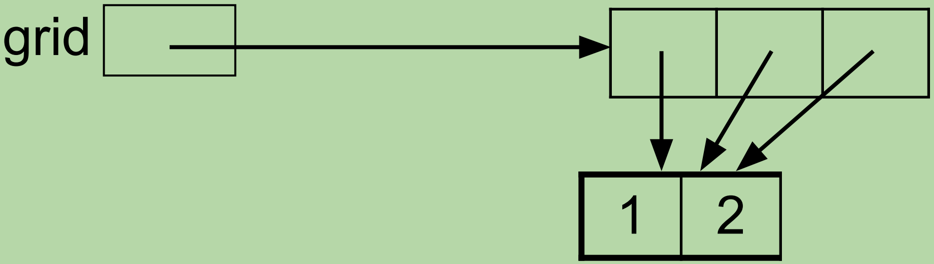


- What will this print?

```
    grid[1][1] = 5
  print(grid)
```

List Multiplication of a 2-D List (cont.)

```
>>> grid = [[1, 2]] * 3
>>> grid
[[1, 2], [1, 2], [1, 2]]
```



- What will this print?

```
    grid[1][1] = 5
  print(grid)            # output: [[1, 5], [1, 5], [1, 5]]
```