References and Mutable Data

Computer Science 111
Boston University

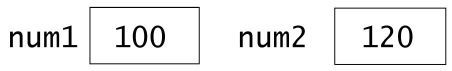
Vahid Azadeh Ranjbar, Ph.D.

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

Recall: Variables as Boxes

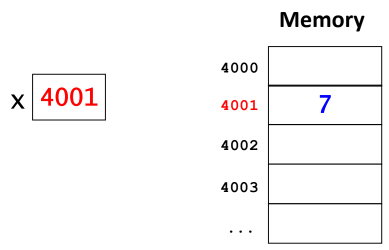
- You can picture a variable as a named "box" in memory.
- Example from an early lecture:

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

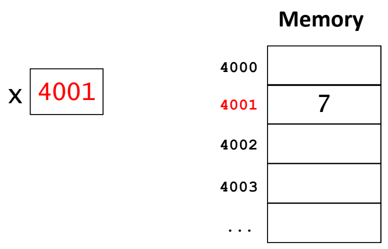


Variables and Values

- 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



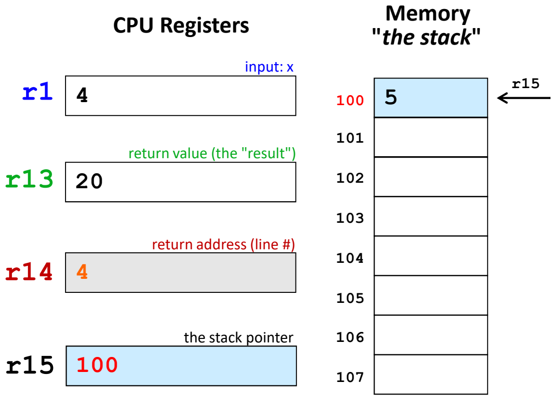
References



- We say that a variable stores a *reference* to its value.
  - also known as a *pointer*
- Where have we seen this before?

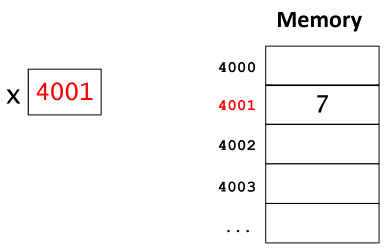
Pointers/References in Assembly

```
       0   read r1
       1   setn r15 100
       2   storer r1 r15
       3   call r14 8
r14 →  4   loadr r1 r15
       5   add r13 r13 r1
       6   write r13
       7   halt
       8   copy r13 r1
       9   addn r1 -1
      10   mul r13 r13 r1
      11   jumpr r14
```

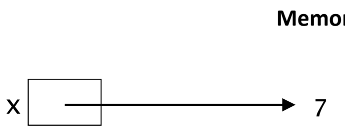


- `r15` (the stack pointer) holds a reference to the value at the top of the stack.
- `r14` (the return address register) holds a reference to the instruction after the function call.

References (cont.)

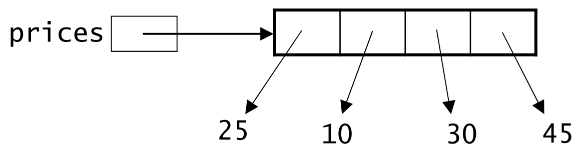


- Because we don't care about the actual memory address, we use an arrow to represent a reference:



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

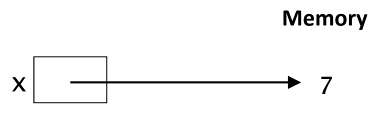
- In Python, strings and numbers are *immutable*.
  - their contents/components cannot be changed
- Lists are *mutable.*
  - their contents/components *can* be changed
  - example:

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

Changing a Value vs. Changing a Variable

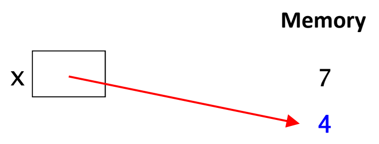
- There's no way to change an immutable value like 7.

`x = 7`



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

`x = 4`

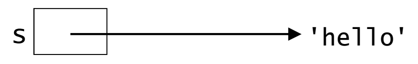


- We're not actually changing the value 7.
- We're making the variable x refer to a different value.

Changing a Value vs. Changing a Variable

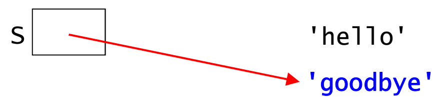
- There's no way to change an immutable *value* like `'hello'`.

`s = 'hello'`



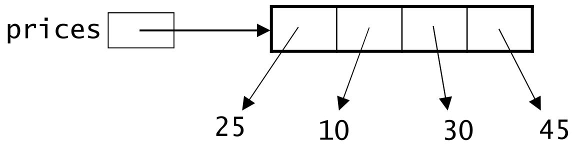
- However, we *can* change the *variable*:

`s = 'goodbye'`



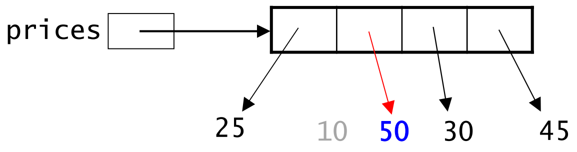
Changing a Value vs. Changing a Variable

- Here's our original list:



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

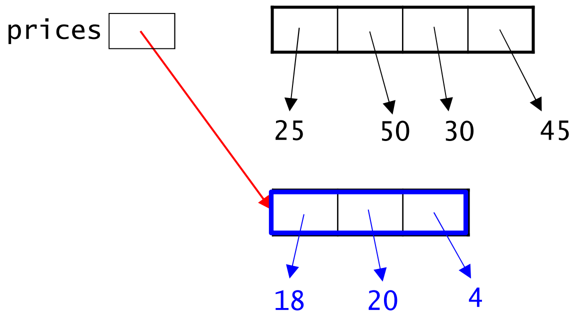
```
prices[1] = 50
```



Changing a Value vs. Changing a Variable

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

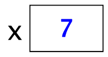
```
prices = [18, 20, 4]
```



Simplifying Our Mental Model

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

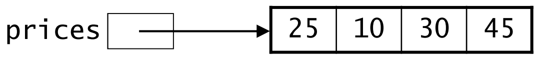
```
x = 7
```



  - a simplified picture, but good enough!

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

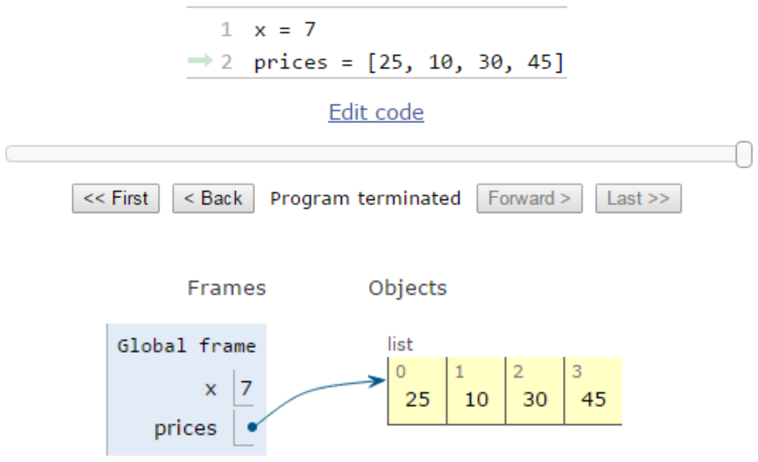
```
prices = [25, 10, 30, 45]
```



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

Simplifying Our Mental Model (cont.)

- Python Tutor uses this simplified model, too:



```
x = 7
prices = [25, 10, 30, 45]
```

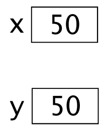
Copying Variables

- The assignment

  var2 = var1

  copies the contents of *var1* into *var2*:

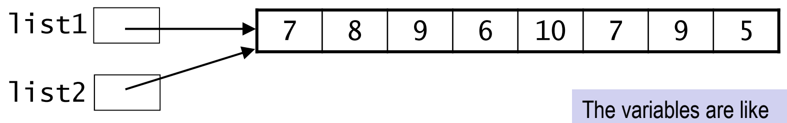
```
x = 50
y = x
```



Copying References

- Consider this example:

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



- Copying a list variable
  simply copies the reference.

- It doesn't copy the list itself!

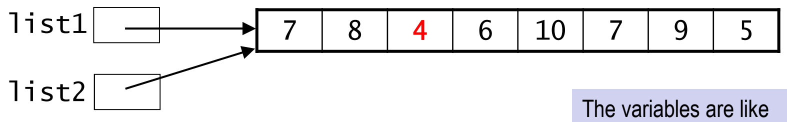
The variables are like two business cards that both have the address of the same office.

The list is the office.

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])
```

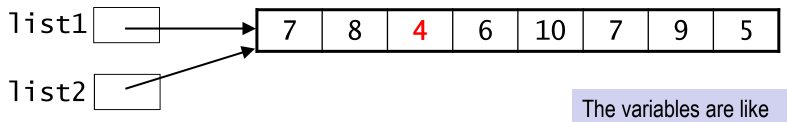
The variables are like two business cards that both have the address of the same office.

The list is the office.

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

The variables are like two business cards that both have the address of the same office.

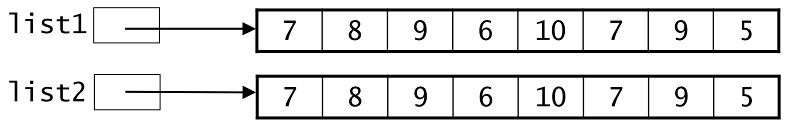
The list is the office.

If you change the contents of the office, someone using either business card to find the office will see the changes!

Copying a List

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

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

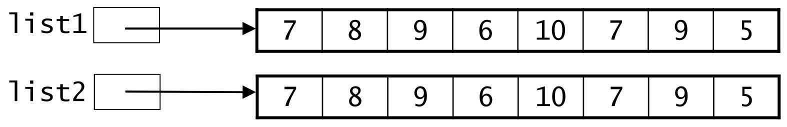


The variables are like business cards for two offices at different addresses. The two offices just happen to have the same contents!

Copying a List

- 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?

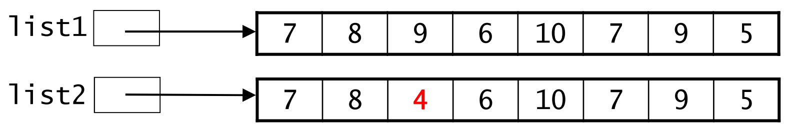
```
list2[2] = 4
print(list1[2], list2[2])
```

The variables are like business cards for two offices at different addresses. The two offices just happen to have the same contents!

Copying a List

- 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
```

The variables are like business cards for two offices at different addresses. The two offices just happen to have the same contents!

Changing the contents of one office doesn't change the other!

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]`

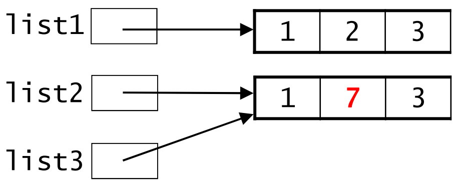
B. `[1, 7, 3] [1, 7, 3] [1, 2, 3]`

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]`

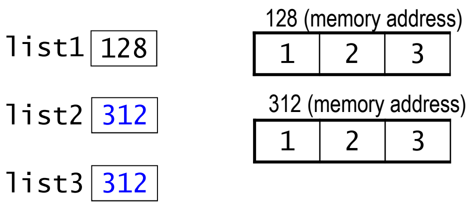
B. `[1, 7, 3] [1, 7, 3] [1, 2, 3]`

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)
```



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

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

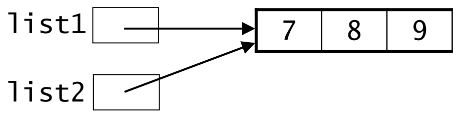
C. `[1, 2, 3] [1, 7, 3] [1, 7, 3]`

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

Changing the Internals vs. Changing a Variable

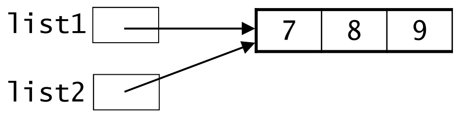
- When two variables hold a reference to the same list...

```
list1 = [7, 8, 9]
list2 = list1
```



- ...if we change *the internals* of the list...

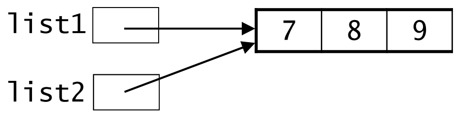
```
list2[2] = 4
print(list1)
```



Changing the Internals vs. Changing a Variable

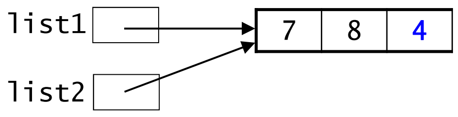
- When two variables hold a reference to the same list...

```
list1 = [7, 8, 9]
list2 = list1
```



- ...if we change *the internals* of the list, both variables will "see" the change:

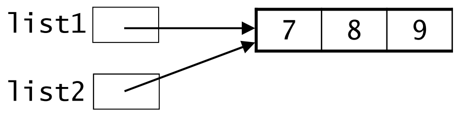
```
list2[2] = 4
print(list1)    # prints [7, 8, 4]
```



Changing the Internals vs. Changing a Variable (cont.)

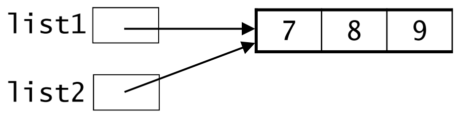
- When two variables hold a reference to the same list...

```
list1 = [7, 8, 9]
list2 = list1
```



- ...if we change one of the variables *itself*...

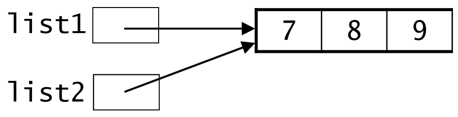
```
list2 = [4, 5, 6]
print(list1)
```



Changing the Internals vs. Changing a Variable (cont.)

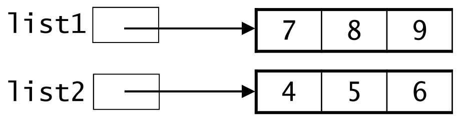
- When two variables hold a reference to the same list...

```
list1 = [7, 8, 9]
list2 = list1
```



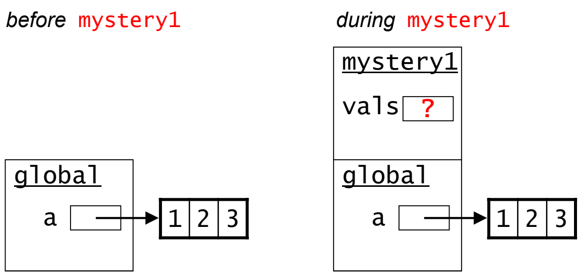
- ...if we change one of the variables *itself*, that does *not* change the other variable:

```
list2 = [4, 5, 6]
print(list1)    # prints [7, 8, 9]
```



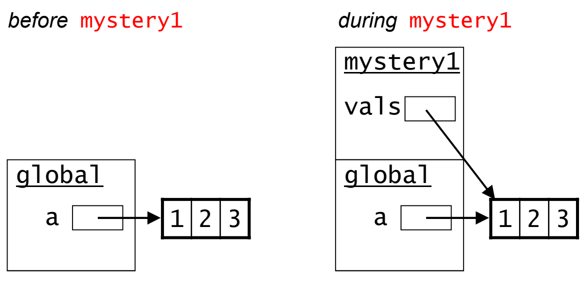
Passing a List to a Function, version 1

```
def mystery1(vals):
    vals[1] = 4

a = [1, 2, 3]
mystery1(a)
print(a)
```



Passing a List to a Function, version 1

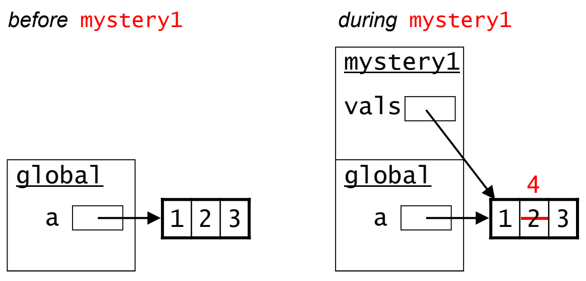
```
def mystery1(vals):
    vals[1] = 4

a = [1, 2, 3]
mystery1(a)
print(a)
```



The function gets a copy of the reference, not a copy of the list!

Passing a List to a Function, version 1

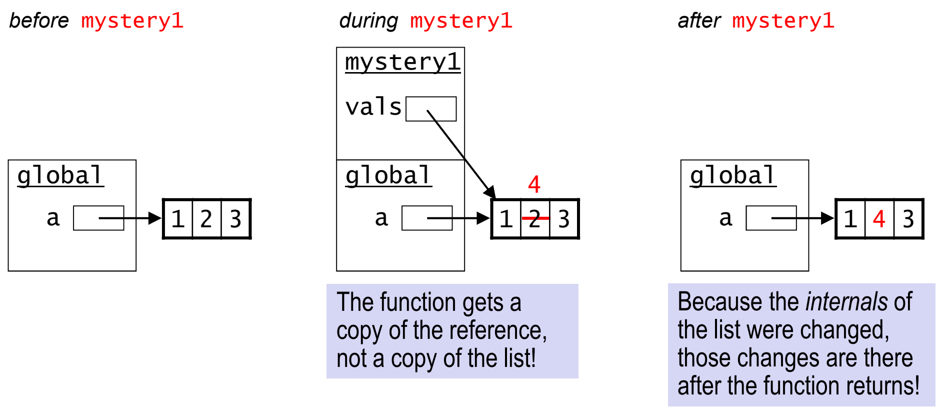
```
def mystery1(vals):
    vals[1] = 4     # changes the internals of the list

a = [1, 2, 3]
mystery1(a)
print(a)
```



The function gets a copy of the reference, not a copy of the list!

Passing a List to a Function, version 1

```
def mystery1(vals):
    vals[1] = 4     # changes the internals of the list

a = [1, 2, 3]
mystery1(a)
print(a)            # prints [1, 4, 3]
```

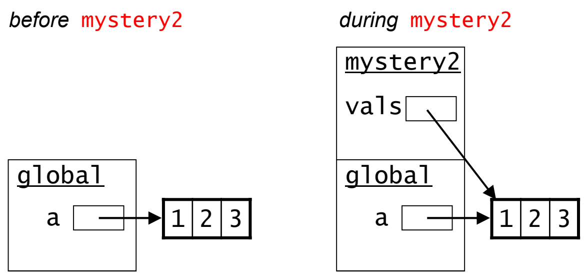


The function gets a copy of the reference, not a copy of the list!

Because the *internals* of the list were changed, those changes are there after the function returns!

Passing a List to a Function, *version 2*

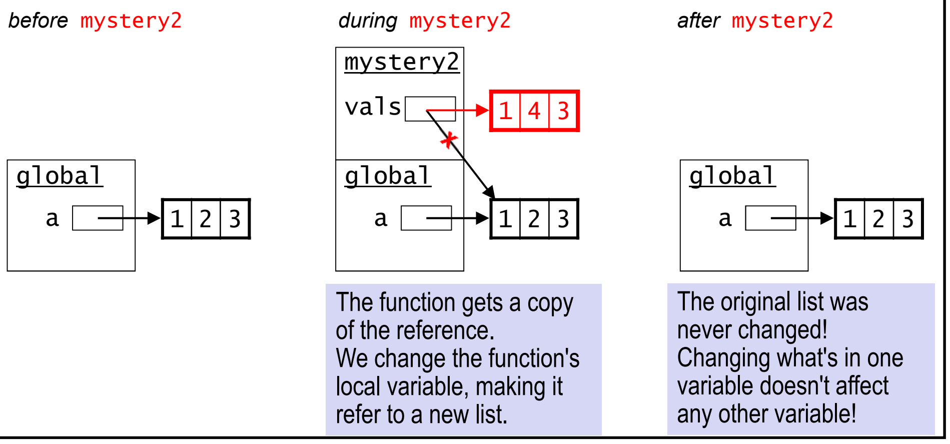
```
def mystery2(vals):
    vals = [1, 4, 3]

a = [1, 2, 3]
mystery2(a)
print(a)
```



The function gets a copy of the reference.

Passing a List to a Function, *version 2*

```
def mystery2(vals):
    vals = [1, 4, 3]    # changes the variable itself

a = [1, 2, 3]
mystery2(a)
print(a)            # prints [1, 2, 3]
```

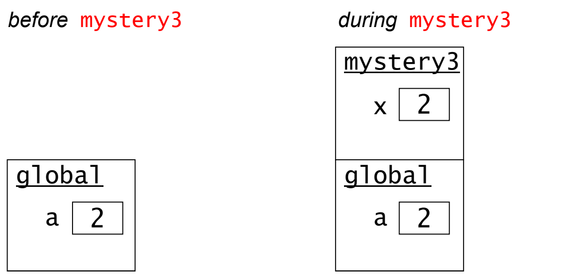


The function gets a copy of the reference.
We change the function's local variable, making it refer to a new list.

The original list was never changed!
Changing what's in one variable doesn't affect any other variable!

Passing an *Immutable* Value to a Function

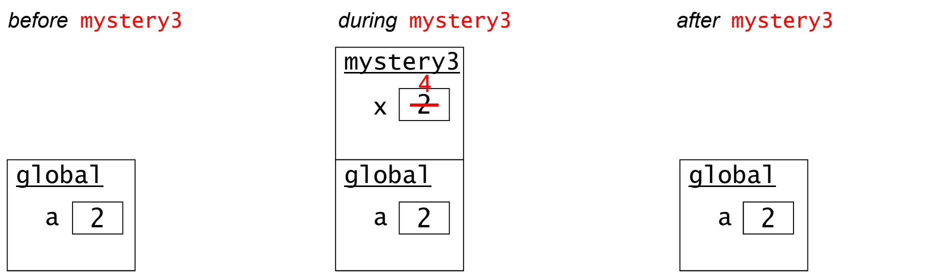
```
def mystery3(x):
    x = x * 2

a = 2
mystery3(a)
print(a)
```



Because the value is immutable, we can think of the function getting a copy of the value.

Passing an *Immutable* Value to a Function

```
def mystery3(x):
    x = x * 2     # changes the variable itself

a = 2
mystery3(a)
print(a)          # prints 2
```



Because the value is immutable, we can think of the function getting a copy of the value.

The global variable has not changed!
Changing what's in one variable doesn't affect any other variable!

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]`

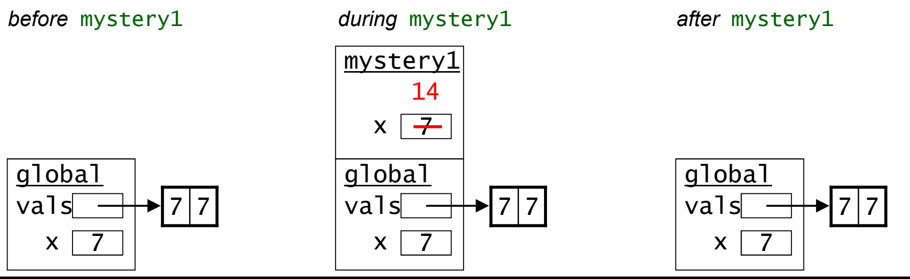
B. `14 [7, 7]`

C. `7 [111, 7]`

D. `14 [111, 7]`

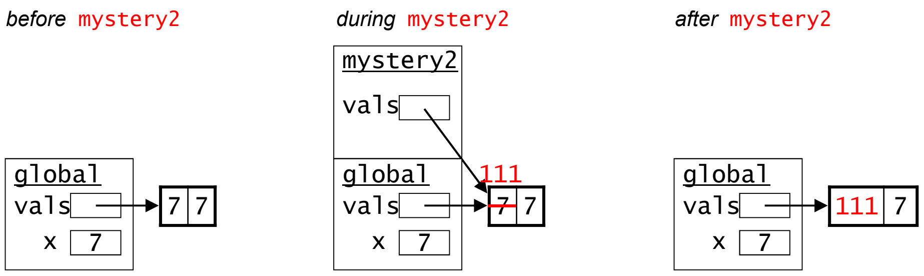
What does this program output?

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

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



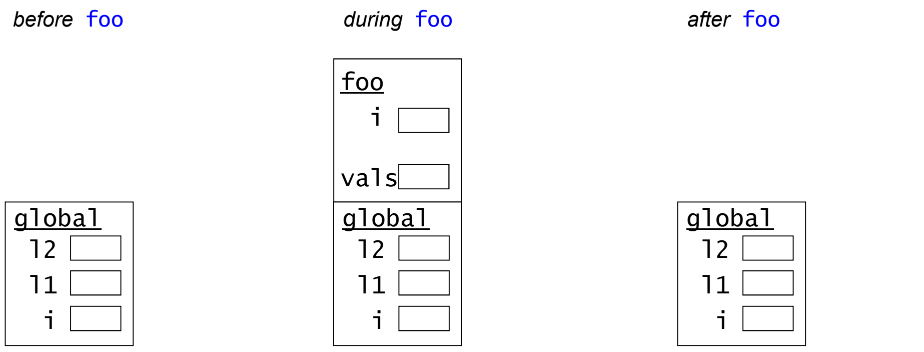
What does this program output?

```
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]
```



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

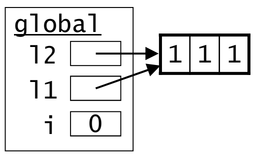
```
def foo(vals, i):
    i += 1
    vals[i] *= 2

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



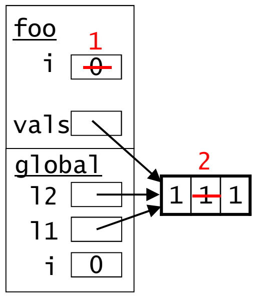
Extra Practice: What does this program print? Draw your own memory diagrams!

```
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]
```

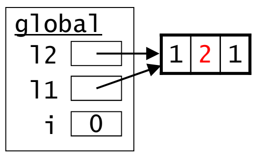
before foo



during foo



after foo



Recall Our Earlier Example...

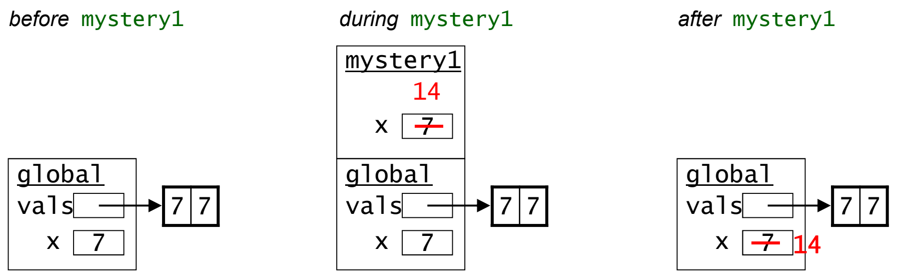
```
def mystery1(x):
    x *= 2
    return x
def mystery2(vals):
    vals[0] = 111
    return vals

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
global
vals
7
7
x
7
during mystery1
mystery1
14
x
~~7~~
global
vals
7
7
x
7
after mystery1
global
vals
7
7
x
~~7~~
14