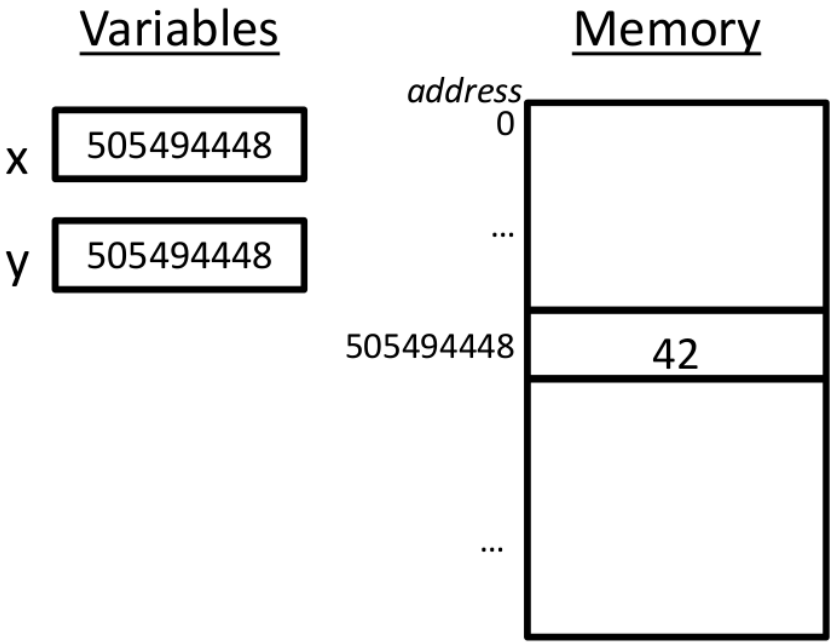
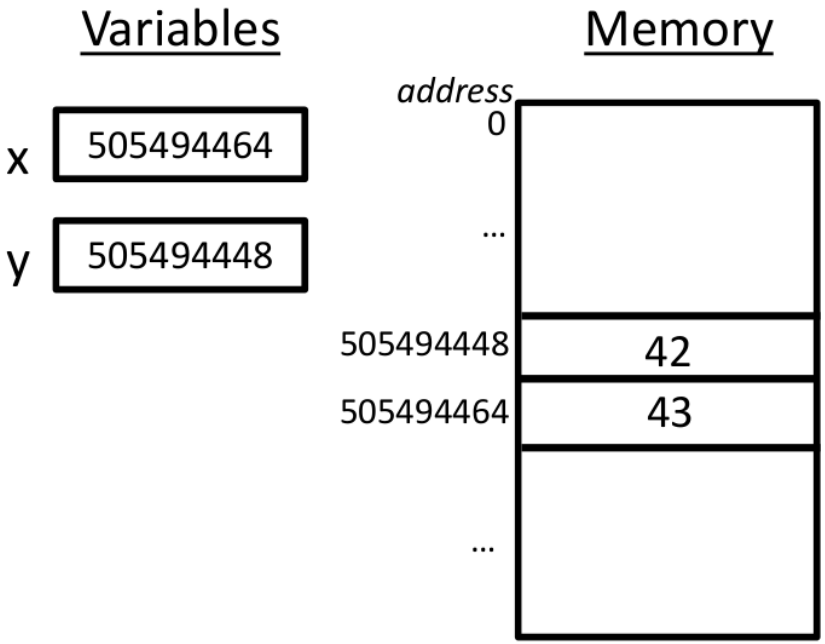
How Python Stores Data

```
x = 42
y = x
```

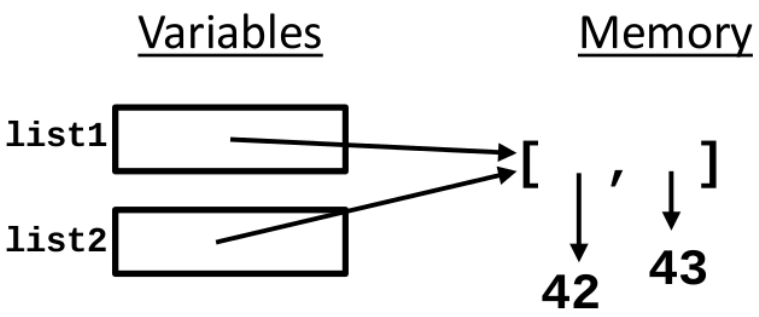



```
x = 42
y = x
x = 43
```

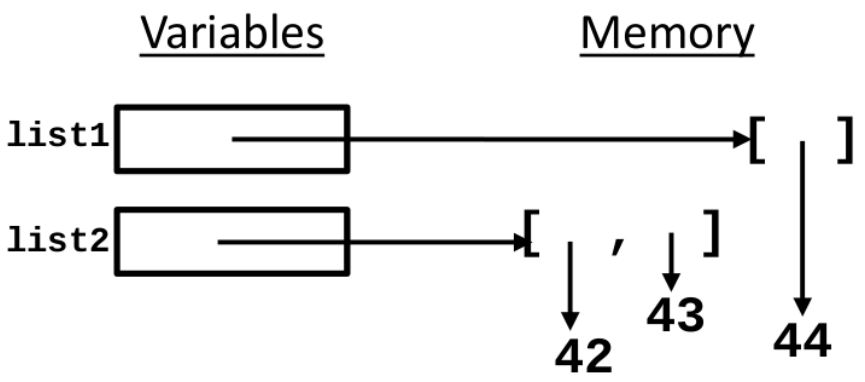


How Python Stores Lists

```
list1 = [42,43]
list2 = list1
```

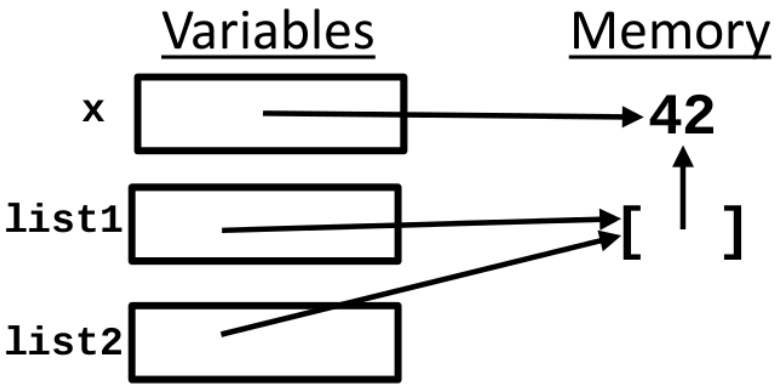



```
list1 = [42,43]
list2 = list1
list1 = [44]
```

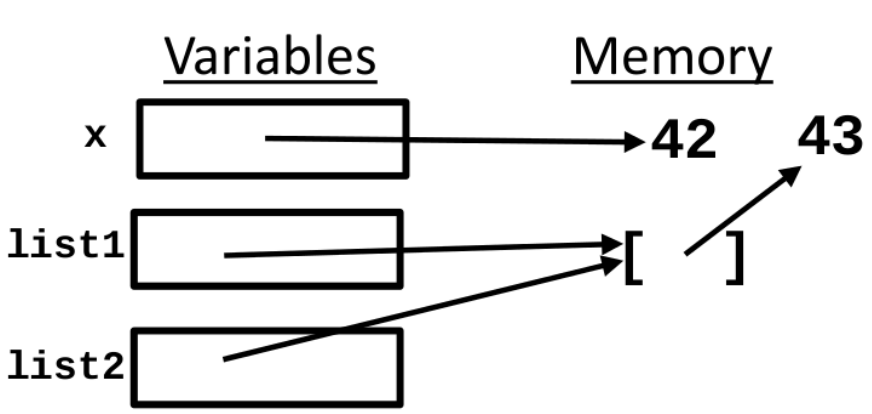


Modifying Elements in a List

```
x = 42
list1 = [x]
list2 = list1
```




```
x = 42
list1 = [x]
list2 = list1
list1[0] = 43
```



(1a) Lists are handled “by **reference**”

In CS,
rules rule!

(1b) Numbers and strings are handled “by **value**”

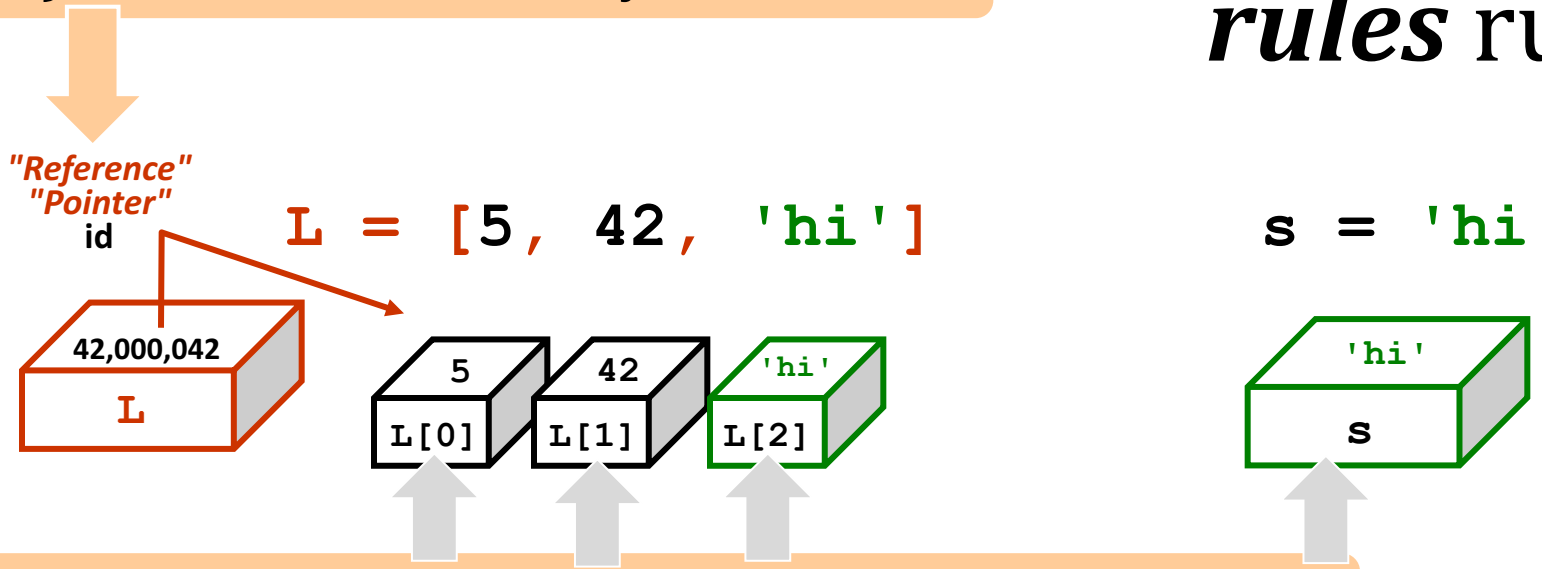
(2) Functions receive arguments “by copy”

two rules,

“interesting” results!

In CS, ***rules*** rule!

(1a) Lists are handled "by **reference**"

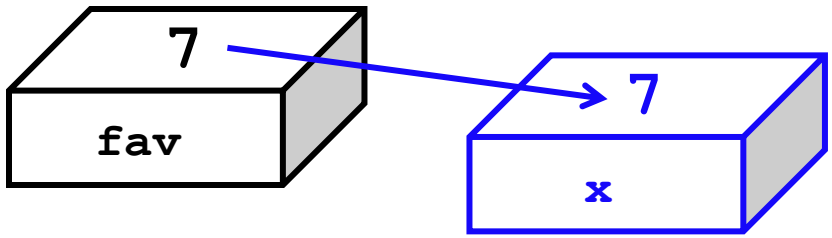


(1b) Numbers and strings are handled "by **value**"

(2) Functions receive inputs "by copy"

The contents of the variable's "box" in memory are copied.

```
in main:
  fav = 7
  f(fav)
```

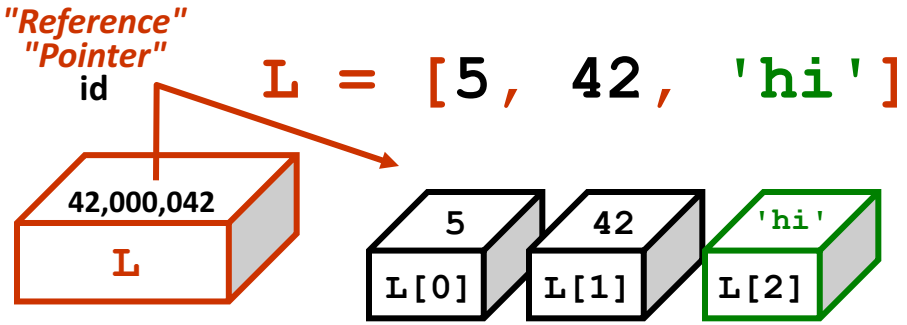



```
def f(x):
```

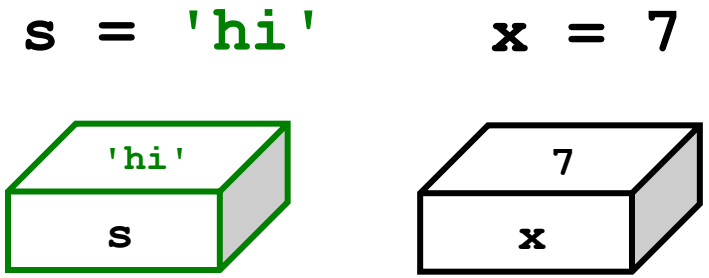
x is a copy of **fav**

Reference vs. Value

Python's two methods for handling data



Lists are handled **by reference**:
`L` really holds a *memory address*



Numeric data and **strings** are handled **by value**: imagine they *hold* the data

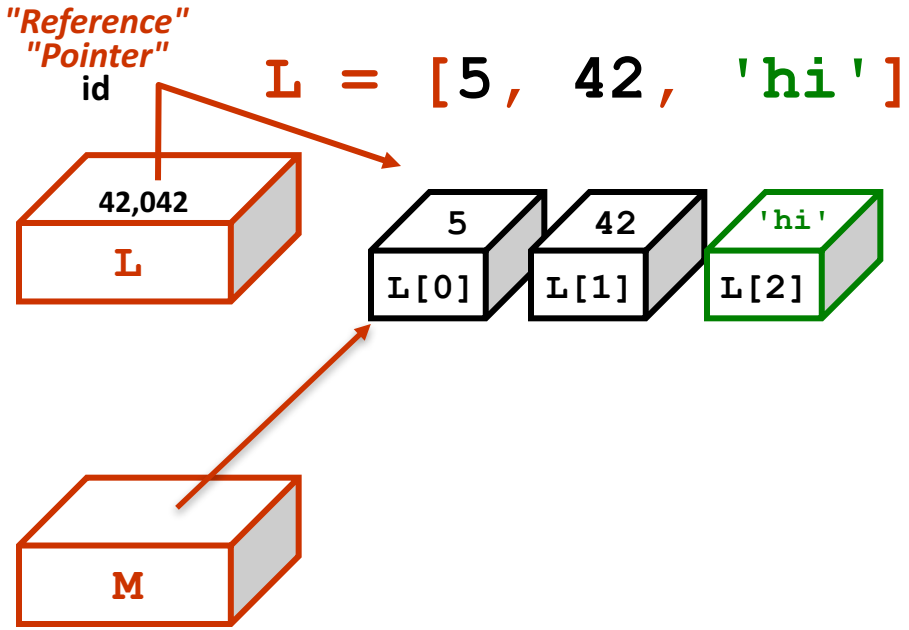
L = [7, 8, 9]; id(L) ...

x = 9001; id(x) ...

s = 'wow!'; id(s) ...

Shallow vs. Deep

Python's two methods for copying data



L = [5, 42, 'hi']

M = L

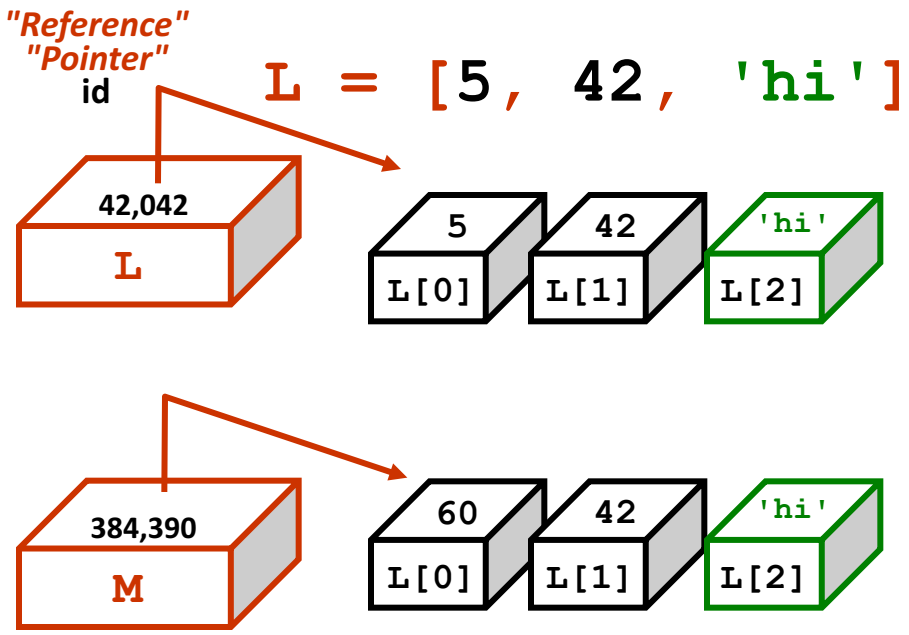
M[0] = 60

What's L[0] ?!

= assignment is "shallow"

Shallow vs. **Deep**

Python's two methods for copying data



from copy import *

L = [5, 42, 'hi']

M = **deepcopy**(L)

M[0] = 60

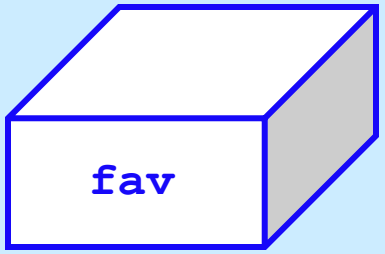
What's L[0] ?!

deepcopy is deep!

Python functions: *pass by copy*

```
def conform(fav)

    fav = 42
    return fav
```

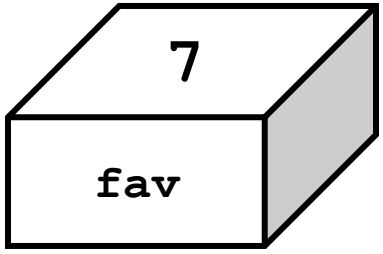



```
def main()

    print("Welcome!")

    fav = 7
    fav = conform(fav)

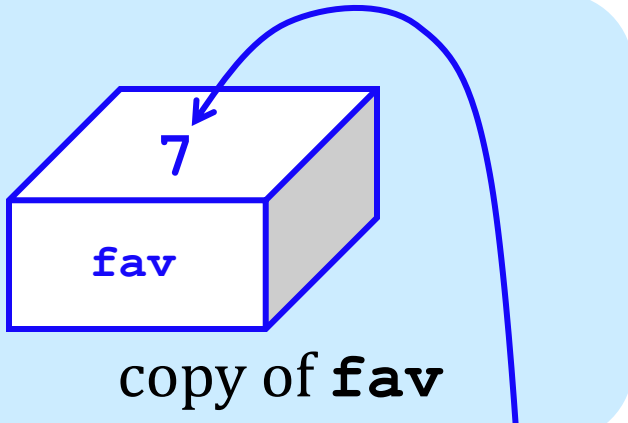
    print("My favorite # is", fav)
```



Python functions: *pass by copy*

```
def conform(fav)

    fav = 42
    return fav
```



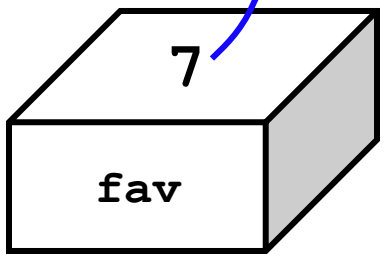
“pass by copy” means the contents of **fav** are copied to **fav**

```
def main()

    print("Welcome!")

    fav = 7
    fav = conform(fav)

    print("My favorite # is", fav)
```



But what if the underlined part were absent… ?

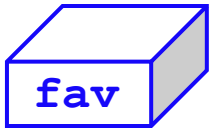
Try it! Rules rule!?

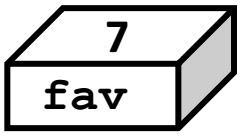
Trace each f'n. What do **`main1`**, **`main2`**, and **`main3`** print?

Numbers: by value. | Lists: by reference. | Function calls *copy*.

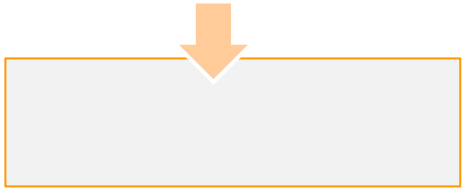
Thought experiments: *Don't hand this in…*

```
def conform1(fav)
    fav = 42
    return fav
```

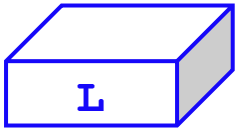


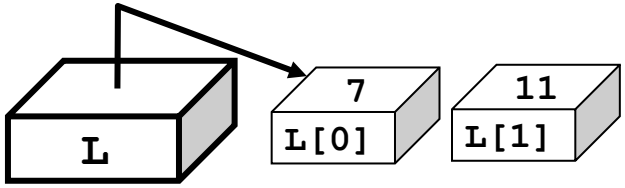



```
def main1()
    fav = 7
    conform1(fav)
    print(fav)
```

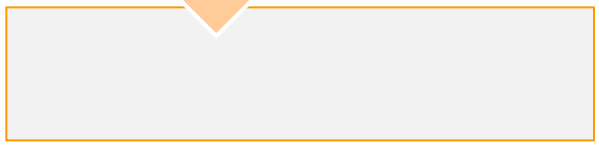


```
def conform2(L)
    L = [42, 42]
    return L
```

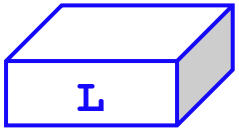


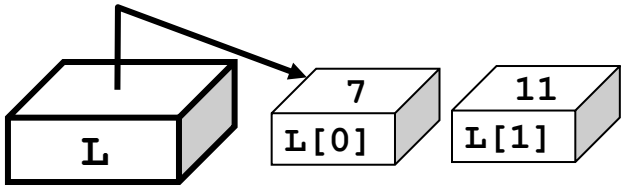



```
def main2()
    L = [7, 11]
    conform2(L)
    print(L)
```



```
def conform3(L)
    L[0] = 42
    L[1] = 42
```






```
def main3()
    L = [7, 11]
    conform3(L)
    print(L)
```



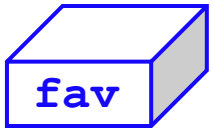
Notice that there are NO assignment statements after these function calls! The return values *aren't being used*…

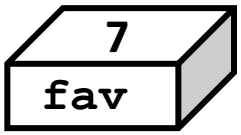
Try it! Rules rule!?

Trace each f'n. What do **`main1`**, **`main2`**, and **`main3`** print?

Numbers: by value. | Lists: by reference. | Function calls *copy*.

```
def conform1(fav)
    fav = 42
    return fav
```

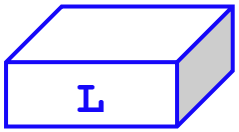


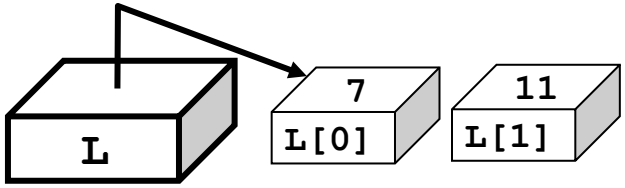



```
def main1()
    fav = 7
    conform1(fav)
    print(fav)
```

7

```
def conform2(L)
    L = [42, 42]
    return L
```

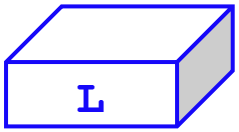


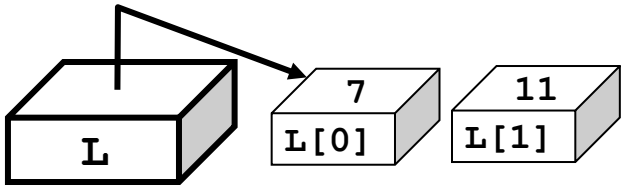



```
def main2()
    L = [7, 11]
    conform2(L)
    print(L)
```

[7, 11]

```
def conform3(L)
    L[0] = 42
    L[1] = 42
```






```
def main3()
    L = [7, 11]
    conform3(L)
    print(L)
```

[42, 42]

Notice that there are NO assignment statements after these function calls! The return values *aren't being used*...

Lists are *Mutable*

You can change **the contents** of lists in functions that take those lists as input.

- Lists are MUTABLE objects

Those changes will be visible **everywhere**.

Numbers, strings, etc. are IMMUTABLE: they can't be changed, only reassigned.

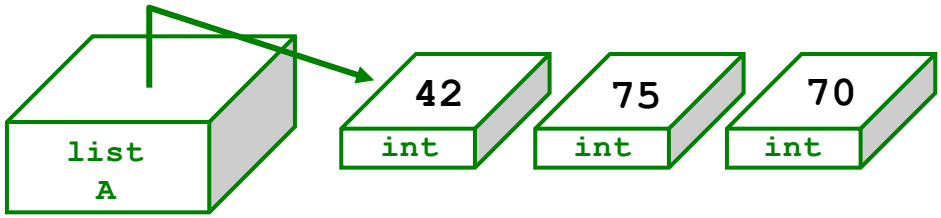
2D data!

All and only the rules that govern 1D data apply here—no new rules to learn!

~ pure composition

Lists ~ 2D data

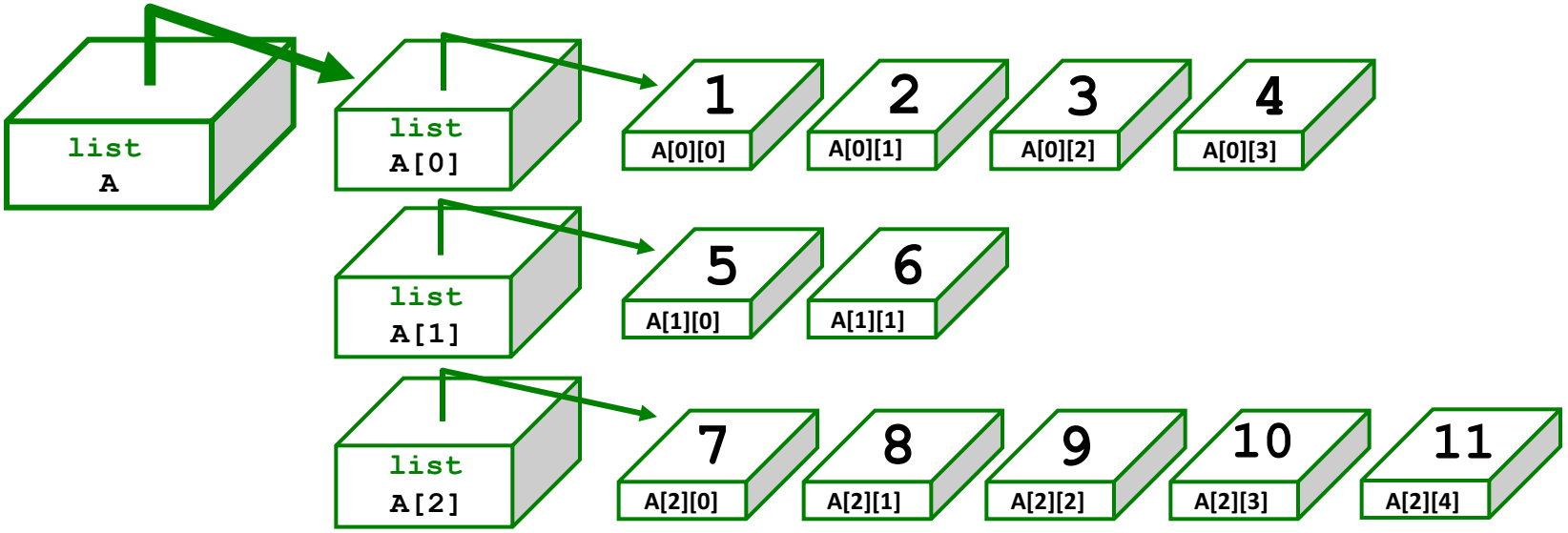
```
A = [42, 75, 70]
```



1D lists are familiar—but lists can hold ANY kind of data—***including lists!***

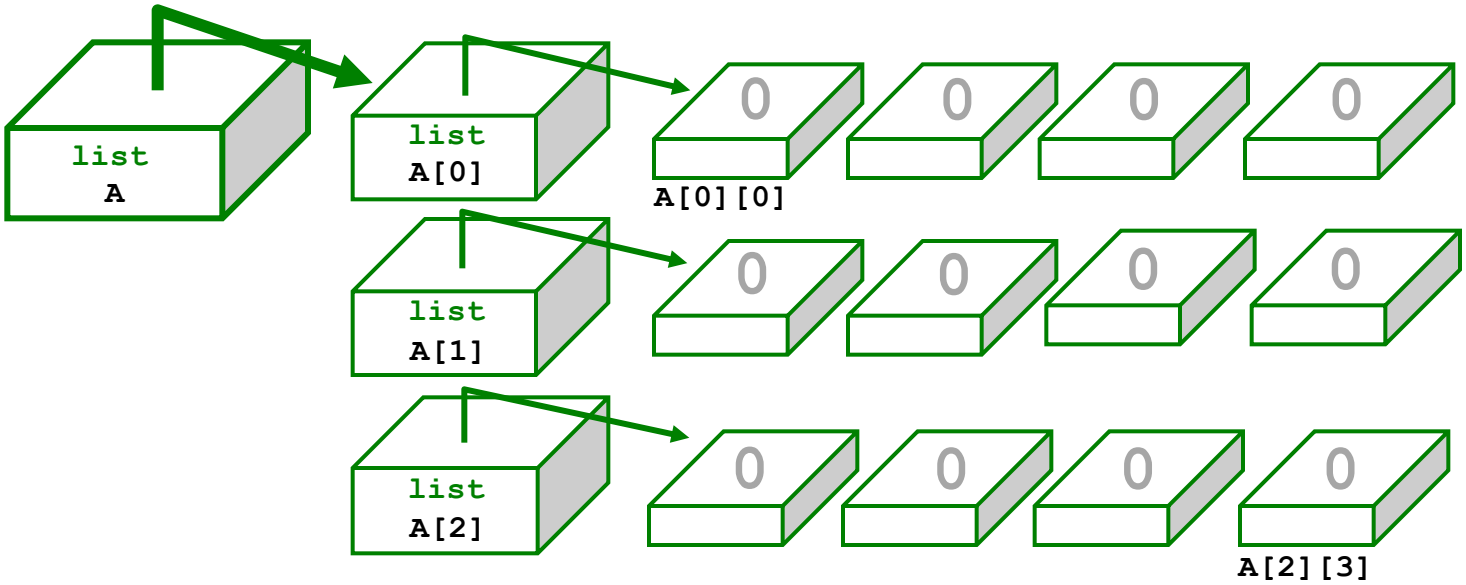
Lists ~ 2D data

```
A = [[1,2,3,4], [5,6], [7,8,9,10,11]]
```



Rectangular 2D data

```
A = [[0,0,0,0], [0,0,0,0], [0,0,0,0]]
```



Original data...

To try…

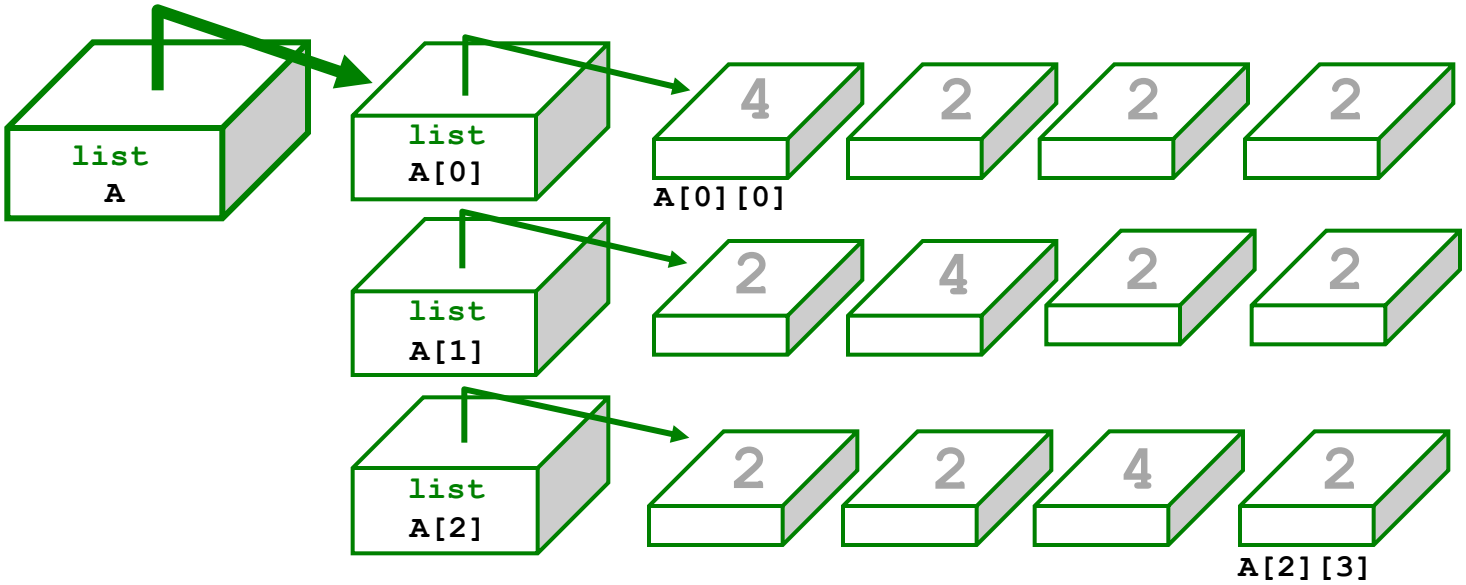
```
NROWS = len(A)
NCOLS = len(A[0])

for r in range(0, NROWS):
    for c in range(0, NCOLS):
        if r == c:    A[r][c] = 4
        else:         A[r][c] = 2
```

What will this do?

Rectangular 2D data

```
A = [[4,2,2,2], [2,4,2,2], [2,2,4,2]]
```



To try…

```
NROWS = len(A)
NCOLS = len(A[0])

for r in range(0, NROWS):
    for c in range(0, NCOLS):
        if r == c:    A[r][c] = 4
        else:         A[r][c] = 2
```

What will this do?

Takeaway Message

- All Python variables have
  - Type (what kind of data is it)
  - Reference (memory location of the value)
- Python assignment only copies the reference
- Lists allow assignment to component parts
  - If you have different variables that refer to the same list, changes to the list in one variable will be seen in the other variables