Week 4

Tuples, Lists and other Iterables

Tuples

Quick Tuple Exercise

```
Code                             Output
tup_a = (10, 11, 12, 13)
print(tup_a)                     (10, 11, 12, 13)
tup_b = (“CS”, 1010)
print(tup_b)                     (‘CS’, 1010)
tup_c = tup_a + tup_b
print(tup_c)                     (10, 11, 12, 13, ‘CS’, 1010)
print(len(tup_c))                6
```

Quick Tuple Exercise

```
tup_a = (10, 11, 12, 13)
tup_b = (“CS”, 1010)
tup_c = tup_a + tup_b
```

Code	**Output**
`print(11 in tup_a)`	`True`
`print(14 in tup_b)`	`False`
`print(“C” in tup_c)`	`False`
`print(tup_b[1])`	`1010`
`tup_d = tup_b[0]*4`	
`print(tup_d)`	`CSCSCSCS`
`print(tup_b[1] * 4)`	`4040`

```
tup_d = “CSCSCSCS”
```

Quick Tuple Exercise

Code	**Output**
`tup_e = tup_d[1:]`	
`print(tup_e)`	SCSCSCS
`tup_f = tup_d[::-1]`	
`print(tup_f)`	SCSCSCSC
`tup_g = tup_d[1:-1:2]`	
`print(tup_g)`	SSS
`tup_h = tup_d[-1:6:-2]`	
`print(tup_h)`	S

Quick Tuple Exercise

```
Code                    Output
tup_i = (1)
print(tup_i)            1
tup_j = (1,)
print(tup_j)            (1,)
print(tup_i * 4)        4
print(tup_j * 4)        (1, 1, 1, 1)
```

Quick Tuple Exercise

```
tup_a = (10, 11, 12, 13)
tup_b = (“CS”, 1010)
tup_c = tup_a + tup_b
tup_e = “SCSCSCS”
```

Code	**Output**
`print(min(tup_a))`	10
`print(max(tup_a))`	13
`print(min(tup_c))`	TypeError
`print(max(tup_c))`	TypeError
`print(min(tup_e))`	C
`print(max(tup_e))`	S

```
tup_b = (“CS”, 1010)
```

Quick Tuple Exercise

Code

```
for i in tup_b:
  print(i)
```

Output

```
CS
1010
```

Quick Tuple Exercise

Code

```
for i in range(5):
  print(i)
```

Output

```
0
1
2
3
4
```

Quick Tuple Exercise

Code

```
for i in range(2,5):
  print(i)
```

Output

```
2
3
4
```

Quick Tuple Exercise

Code

```
for i in range(2,5,2):
  print(i)
```

Output

```
2
4
```

Quick Tuple Exercise

Code

```
for i in range(5,1,-1):
  print(i)
```

Output

```
5
4
3
2
```

Quick Tuple Exercise

Code

```
for i in range(5,6,-1):
  print(i)
```

Output

Tuple

- Definition
    - *Immutable* sequence of Python objects
    - Enclosed in parentheses
    - Objects are separated by commas
- Operations
    - `len(x)` returns the number of elements in `x`
    - `elem in x` returns `True` if `elem` is in `x`, and `False` otherwise
    - `for var in x` iterates over the elements of `x`; each element is stored in `var`
    - `max(x)` returns the maximum element in `x`
    - `min(x)` returns the minimum element in `x`

Tuple Access

- To retrieve an element in a tuple, you use square brackets [ ]
- There are two types of indices for a tuple of size n
  - Forward index starts from 0, ends at n-1
  - Backward index starts from -1, ends at –n

forward	0	1	2	3	4	5	6	7	
	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	
	-8	-7	-6	-5	-4	-3	-2	-1	backward

- Example
  - Let `tup = (1, 2, 3, (4, 5), 6, 7)`
  - `tup[2] 3`
  - `tup[-2]   6`
  - `tup[3] (4, 5)`

Tuple Access Exercises

- For each of the following tuples, write an expression that will return the value 4 from within the tuple
  - `tup1 = (1, 2, 3, 4, 5, 6, 7, 8)`
  - `tup2 = (1, (2, 3, 4), (5, 6, 7), (8,))`
  - `tup3 = (1, (2, 3, (4,), 5), (6, 7, 8))`
- What is the length of each tuple?
- Which of the following will evaluate to `True`?
  - `4 in tup1`
  - `4 in tup2`
  - `4 in tup3`

Lists

Quick List Exercise

Code **Output**

```
lst_a = [“CS”, 1010]
print(lst_a)                [‘CS’, 1010]
lst_b = [“E”,(“is”, “easy”)]
print(lst_b)                [‘E’,(‘is’, ‘easy’)]
lst_c = lst_a + lst_b
print(lst_c)                [‘CS’, 1010, ‘E’, (‘is’, ‘easy’)]
```

```
lst_a = [“CS”, 1010]
```

Quick List Exercise

Code

```
tup_a = (“CS”, 1010)
tup_a[1] = 2030
lst_a[1] = 2030
print(lst_a)
```

Output

```

TypeError

[‘CS’, 2030]
```

```
lst_a = [“CS”, 2030]
```

Quick List Exercise

Code

```
lst_a.append(“E”)
print(lst_a)
lst_a.extend(“easy”)
print(lst_a)
```

Output

```
[‘CS’, 2030, ‘E’]

[‘CS’, 2030, ‘E’, ‘e’, ‘a’,
 ‘s’, ‘y’]
```

```
lst_b = [“E”,(“is”, “easy”)]
```

Quick List Exercise

```
Code                          Output
cpy_b = lst_b[:]
print(cpy_b)                  [‘E’, (‘is’, ‘easy’)]
cpy_b[1] = “is hard”
print(cpy_b)                  [‘E’, ‘is hard’]
print(lst_b)                  [‘E’, (‘is’, ‘easy’)]
```

Quick List Exercise

Code **Output**

```
lst_d = [1, [2], 3]
cpy_d = lst_d[:]
print(cpy_d)                   [1, [2], 3]
print(lst_d)                   [1, [2], 3]
lst_d[1][0] = 9
print(cpy_d)                   [1, [9], 3]
print(lst_d)                   [1, [9], 3]
```

Quick List Exercise

```
lst_d = [1, [9], 3]
cpy_d = [1, [9], 3]
```

Code	**Output**
`print(lst_d == cpy_d)`	`True`
`print(lst_d is cpy_d)`	`False`
`print(lst_d[1] == cpy_d[1])`	`True`
`print(lst_d[1] is cpy_d[1])`	`True`

List

- Definition
  - ***Mutable*** sequence of Python objects
  - Enclosed in square brackets
  - Objects are separated by commas
- Operations
  - `len(x)` returns the number of elements in `x`
  - `elem in x` returns `True` if `elem` is in `x`, and `False` otherwise
  - `for var in x` iterates over the elements of `x`; each element is stored in `var`
  - `max(x)` returns the maximum element in `x`
  - `min(x)` returns the minimum element in `x`

List

Given a list `lst`:

- Mutation

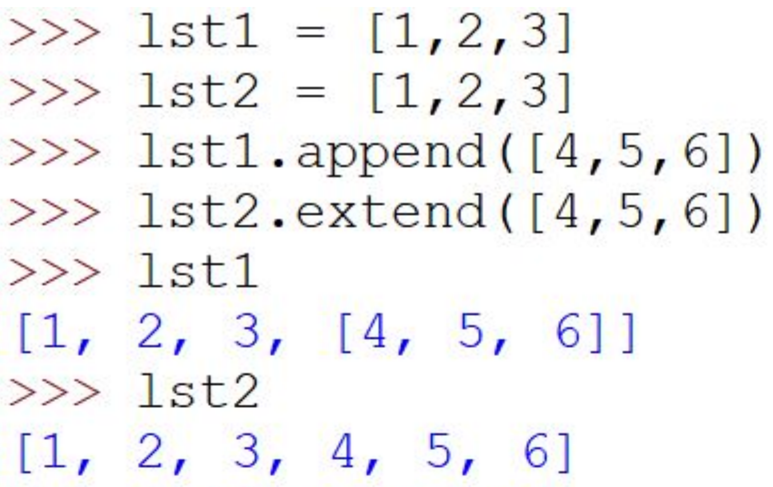
  Changing (the original) lst

  - `lst.append(x)` adds element `x` (*no return*)
  - `lst.extend(x)` adds elements of the iterable `x` (*no return*)
  - `lst.reverse()` reverses `lst` (*no return*)
  - `lst.insert(i,x)` inserts element `x` at index `i`
  - `lst.pop()` removes and returns the last element of `lst`
  - `lst.pop(i)` removes and returns the element of `lst` at index `i`
  - `lst.remove(x)` removes the first occurrence of `x`
  - `lst.clear()` empties the contents of `lst`
- Copy
  - `lst.copy()` returns a *shallow* copy of `lst`

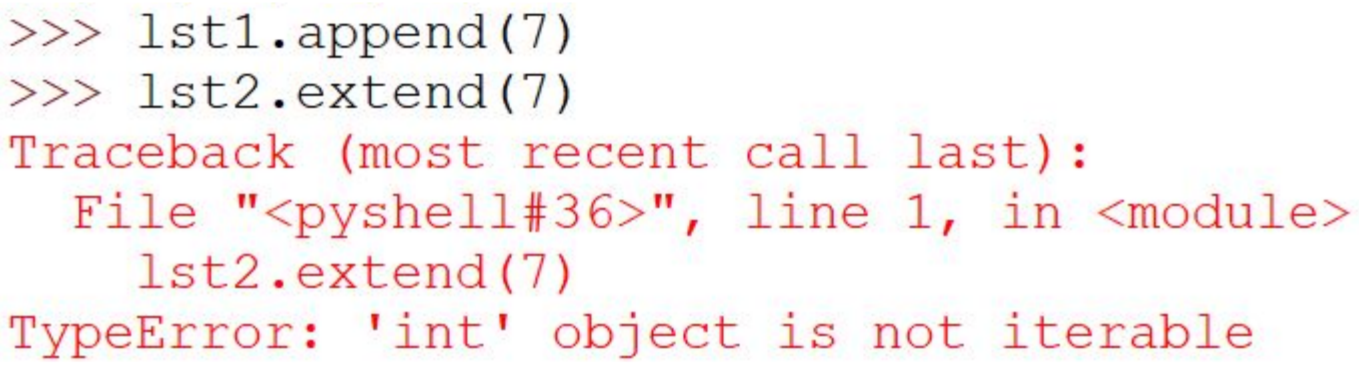
List

Given a list `lst`:

- Mutation
  - `lst.append(x)` adds element x (*no return*)
  - `lst.extend(x)` adds elements of the iterable x (*no return*)
- Difference?



```
>>> lst1 = [1,2,3]
>>> lst2 = [1,2,3]
>>> lst1.append([4,5,6])
>>> lst2.extend([4,5,6])
>>> lst1
[1, 2, 3, [4, 5, 6]]
>>> lst2
[1, 2, 3, 4, 5, 6]
```



```
>>> lst1.append(7)
>>> lst2.extend(7)
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    lst2.extend(7)
TypeError: 'int' object is not iterable
```

List Access

- To retrieve an element in a list, you use square brackets [ ]
- There are two types of indices for a list of size n
  - Forward index starts from 0, ends at n-1
  - Backward index starts from -1, ends at –n

forward	0	1	2	3	4	5	6	7	
	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	
	-8	-7	-6	-5	-4	-3	-2	-1	backward

- Example
  - Let lst = [1, 2, 3, [4, 5], 6, 7]
  - lst[2] 3
  - lst[-2] 6
  - lst[3] [4, 5]

Remember the THREE Types of Loops?

A. Must run exactly N times (definite)

B. Run any number of times (indefinite)

C. Run at most N times (definite loop that may break)

   - Check all True (or check all False)
   - Find any True (or False)

Which Type is it?

- Given a list of N numbers
  - Sum them
  - Calculate the mean/standard deviation
  - Find the max/min
  - Etc.

[✓ A. Must run exactly N times (definite)]

B. Run any number of times (indefinite)

C. Run at most N times (definite loop that may break)

- Check all True (or check all False)
- Find any True (or False)

Which Type is it?

- Given a list
  - Search for a certain object
  - Check if the objects satisfy certain properties, e.g.
    - all odd numbers
    - all strings
    - there exists some abnormal objects

A. Must run exactly N times (definite)

B. Run any number of times (indefinite)

C. Run at most N times (definite loop that may break)

- Check all True (or check all False)
- Find any True (or False)

How about B?

- Think of any example?
  - From user/file input, put data into a list
  - …

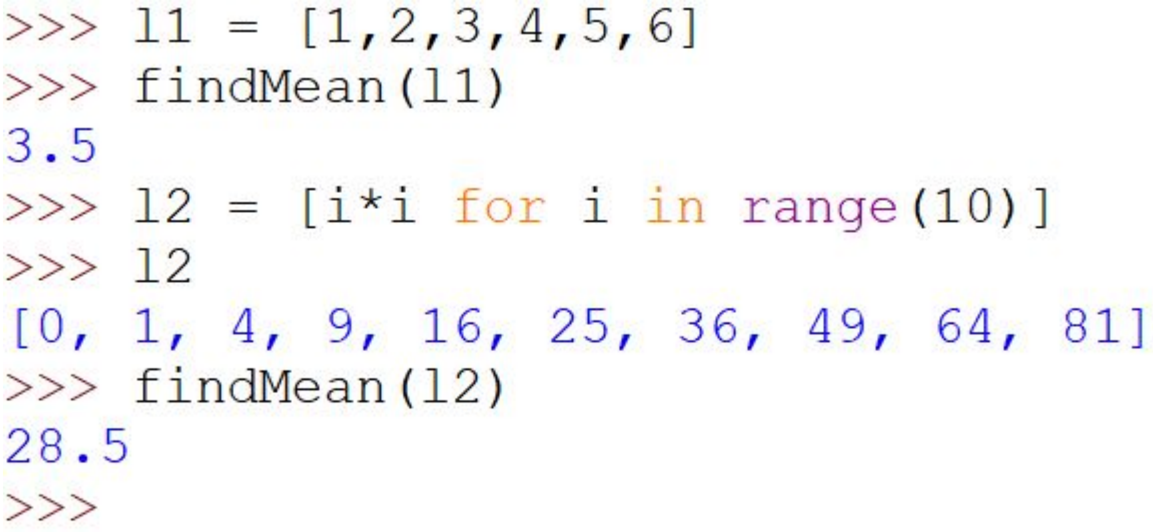
A. Must run exactly N times (definite)

B. Run any number of times (indefinite)

C. Run at most N times (definite loop that may break)
  - Check all True (or check all False)
  - Find any True (or False)

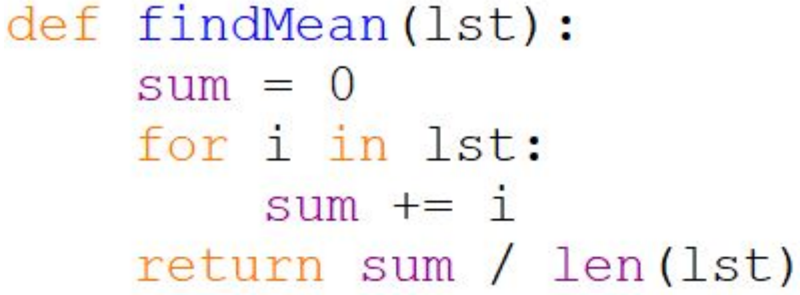
Example #1

Implement `findMean(lst)` where `lst` is a list of numbers.



```
>>> l1 = [1,2,3,4,5,6]
>>> findMean(l1)
3.5
>>> l2 = [i*i for i in range(10)]
>>> l2
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> findMean(l2)
28.5
>>> 
```

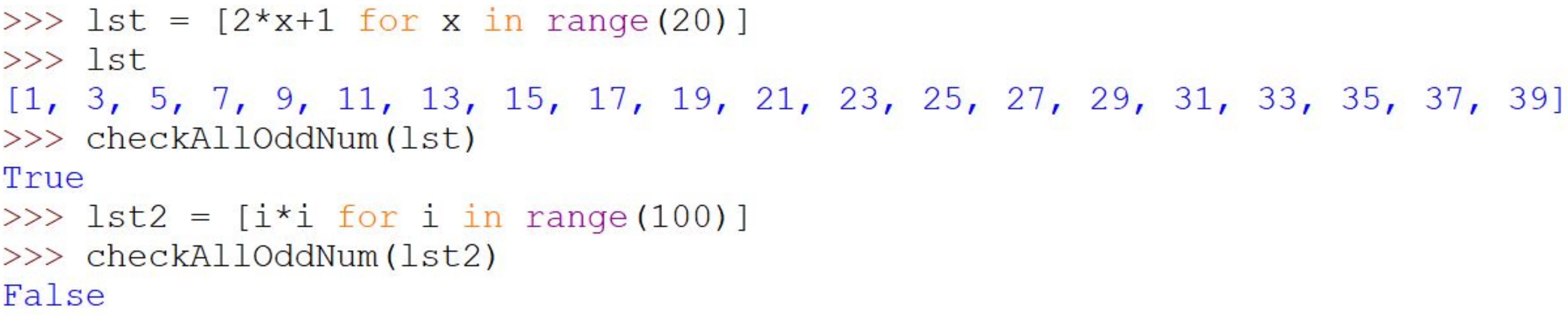
Try it yourself first. (5 min)



```
def findMean(lst):
    sum = 0
    for i in lst:
        sum += i
    return sum / len(lst)
```

Example #2

Implement `checkAllOddNum(lst)` where `lst` is a list of numbers.



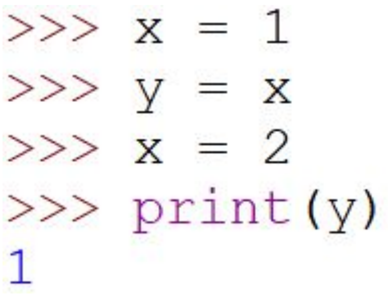
```
>>> lst = [2*x+1 for x in range(20)]
>>> lst
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39]
>>> checkAllOddNum(lst)
True
>>> lst2 = [i*i for i in range(100)]
>>> checkAllOddNum(lst2)
False
```

- Try it yourself. (10 min)

Mutation

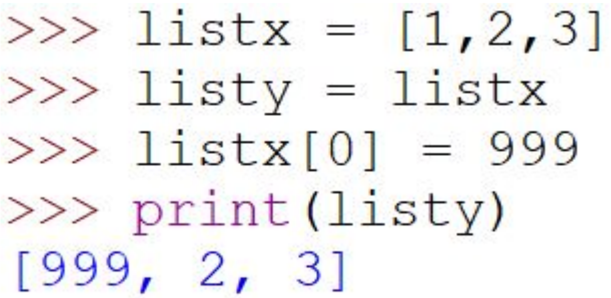
Example #1

<u>Primitive Data Types</u>



```
>>> x = 1
>>> y = x
>>> x = 2
>>> print(y)
1
```

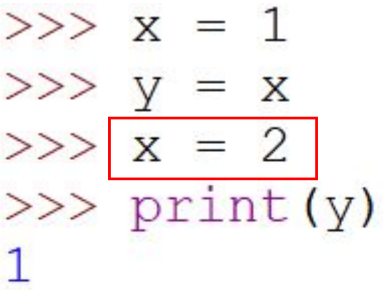
<u>Lists</u>



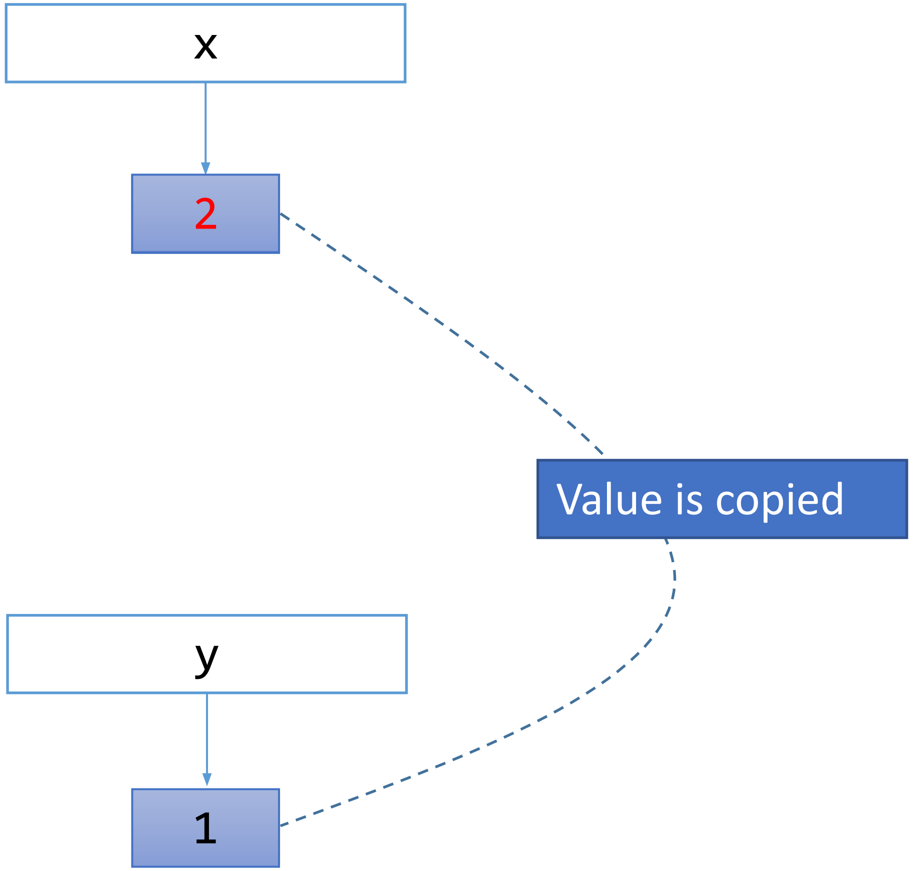
```
>>> listx = [1,2,3]
>>> listy = listx
>>> listx[0] = 999
>>> print(listy)
[999, 2, 3]
```

- We change `listx`
  - But `listy` is also changed?

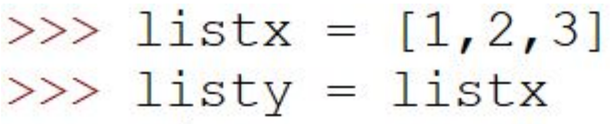
Primitive Data Types



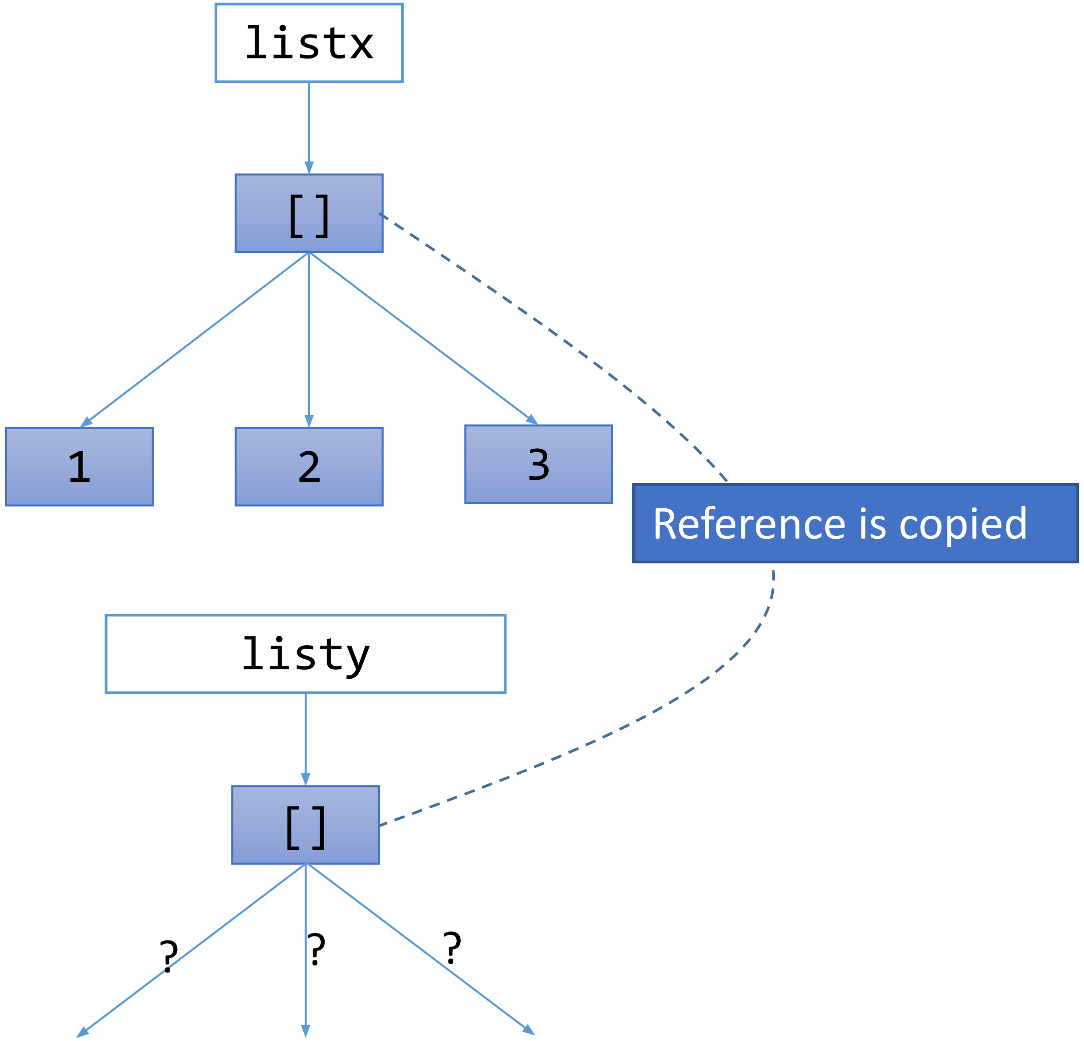
```
>>> x = 1
>>> y = x
>>> x = 2
>>> print(y)
1
```



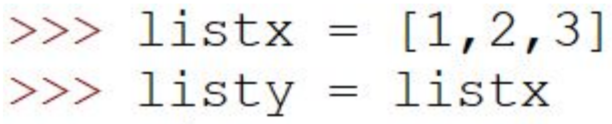
Lists



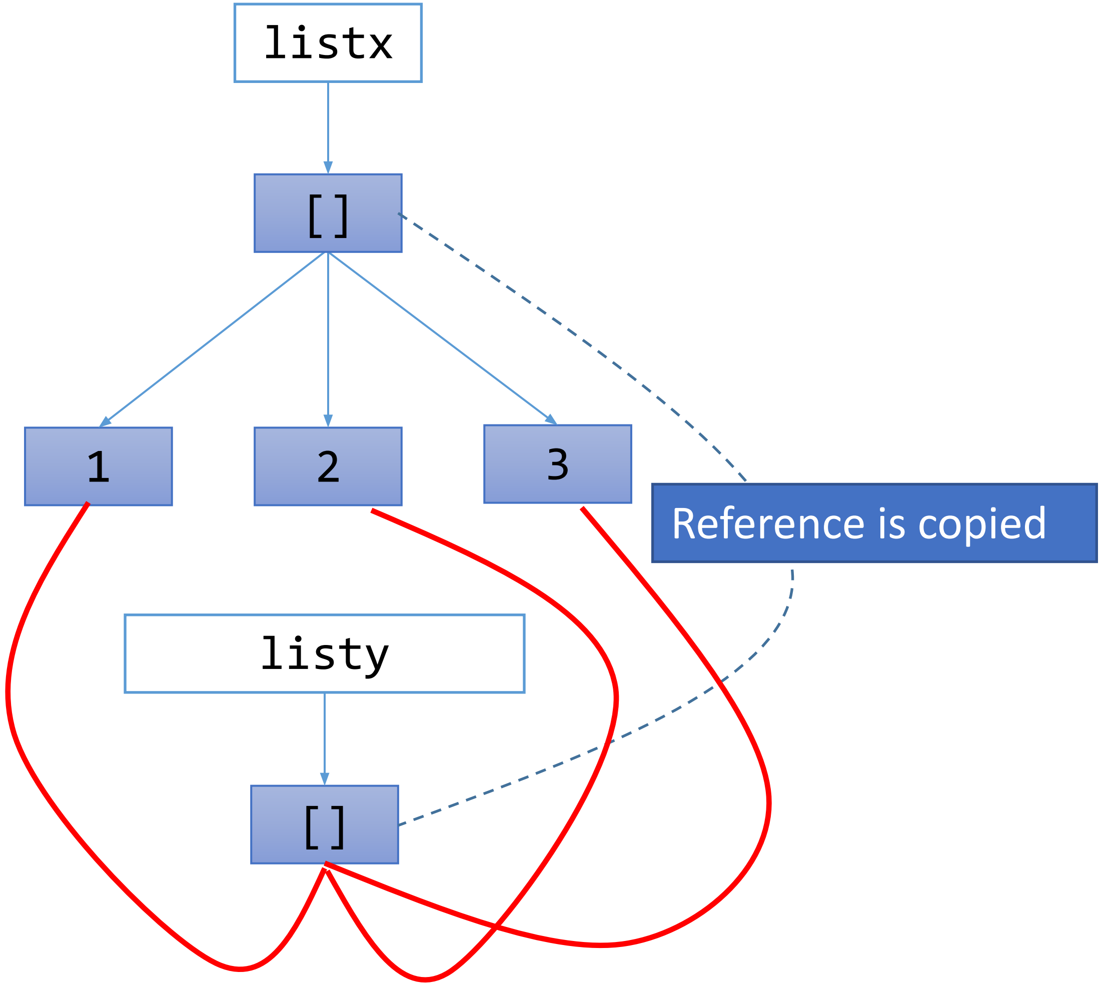
```
>>> listx = [1,2,3]
>>> listy = listx
```



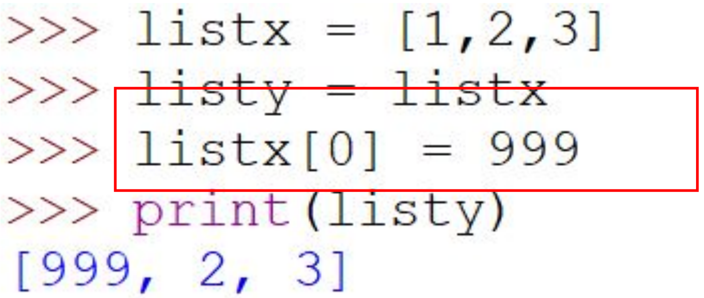
Lists



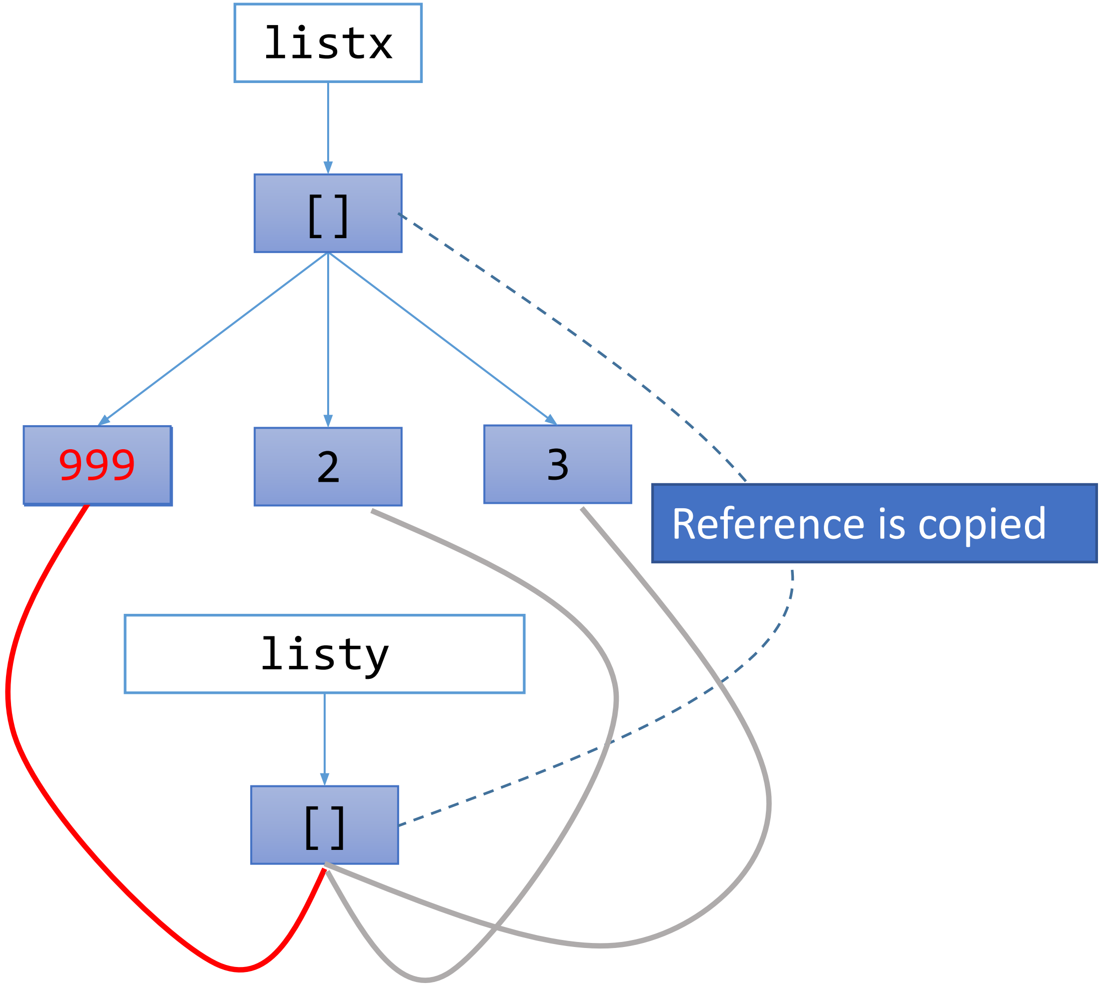
```
>>> listx = [1,2,3]
>>> listy = listx
```



Lists

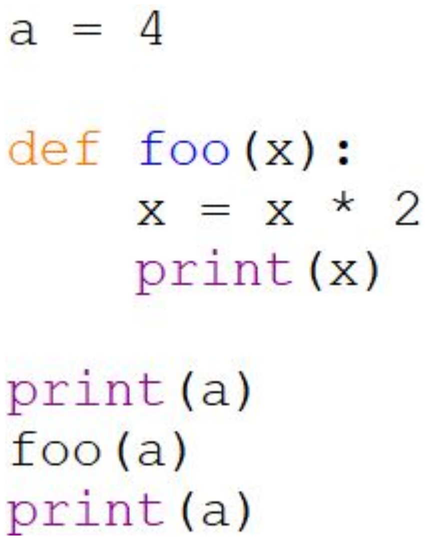


```
>>> listx = [1,2,3]
>>> listy = listx
>>> listx[0] = 999
>>> print(listy)
[999, 2, 3]
```

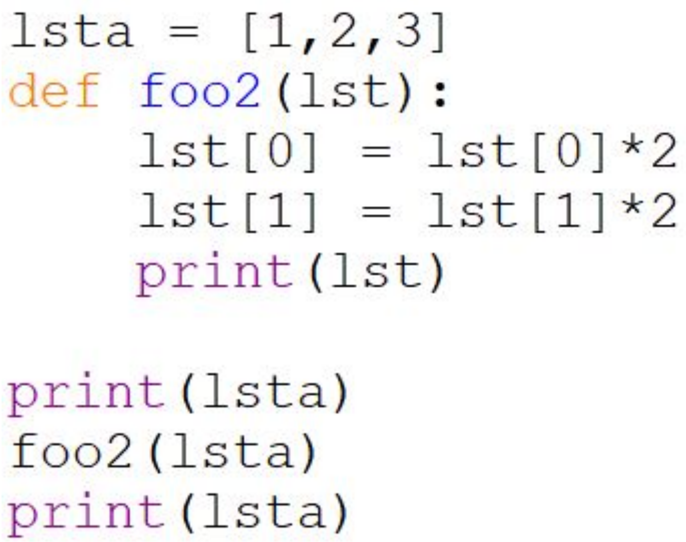


Reference is copied

Example #2



```
a = 4

def foo(x):
    x = x * 2
    print(x)

print(a)
foo(a)
print(a)
```



```
lsta = [1,2,3]
def foo2(lst):
    lst[0] = lst[0]*2
    lst[1] = lst[1]*2
    print(lst)

print(lsta)
foo2(lsta)
print(lsta)
```

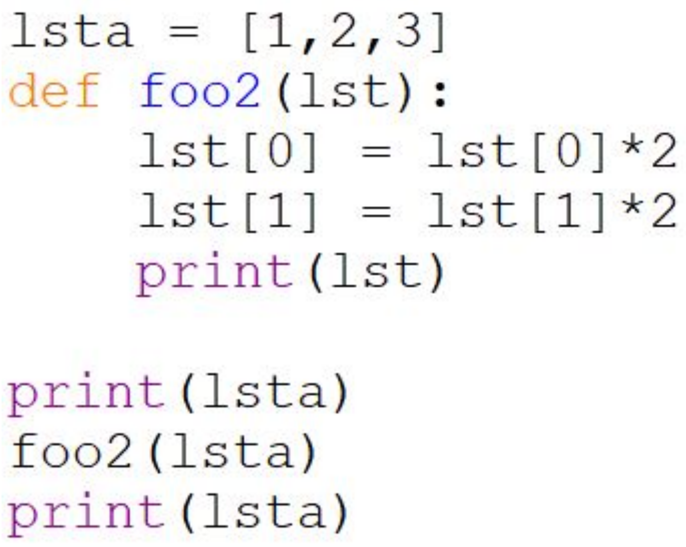
```
4
8
4
```

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

Note the difference

Example #2

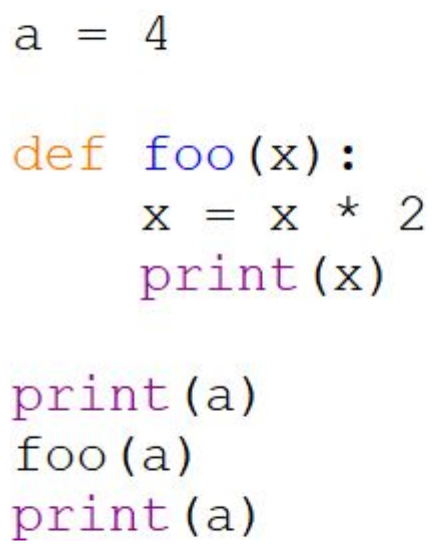
- Unlike “pass by value”
- It is possible to “mutate” a list that is passed into a function



```
lsta = [1,2,3]
def foo2(lst):
    lst[0] = lst[0]*2
    lst[1] = lst[1]*2
    print(lst)

print(lsta)
foo2(lsta)
print(lsta)
```

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

Copy what?

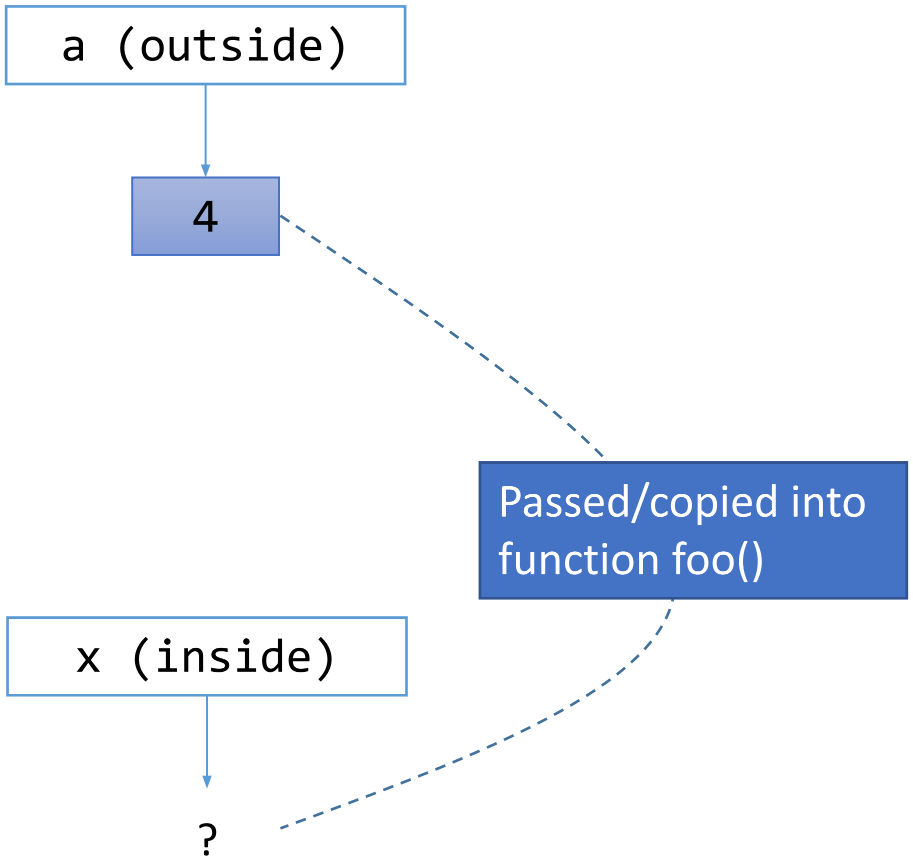


```
a = 4

def foo(x):
    x = x * 2
    print(x)

print(a)
foo(a)
print(a)
```

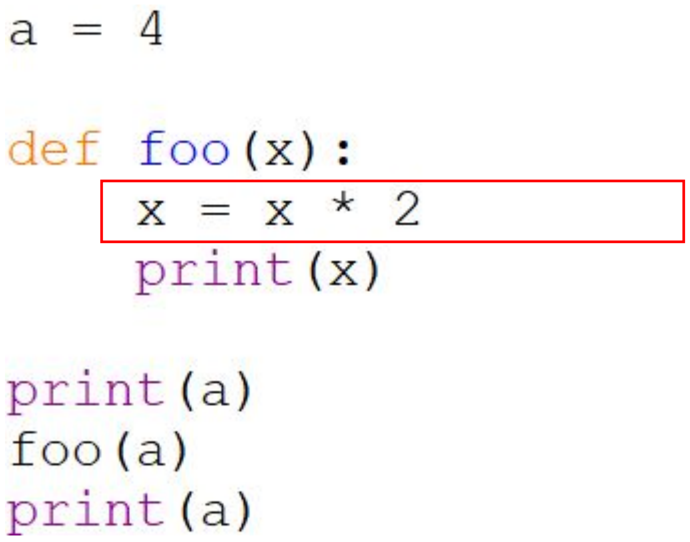
4

8

4

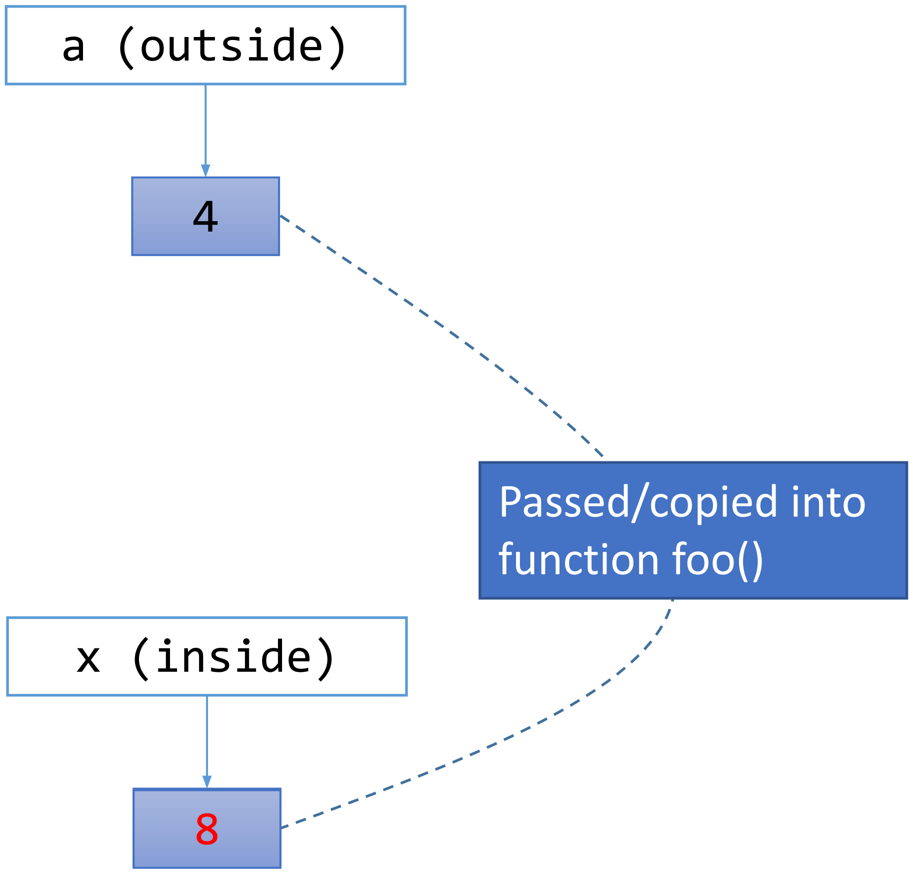


Copy the VALUE

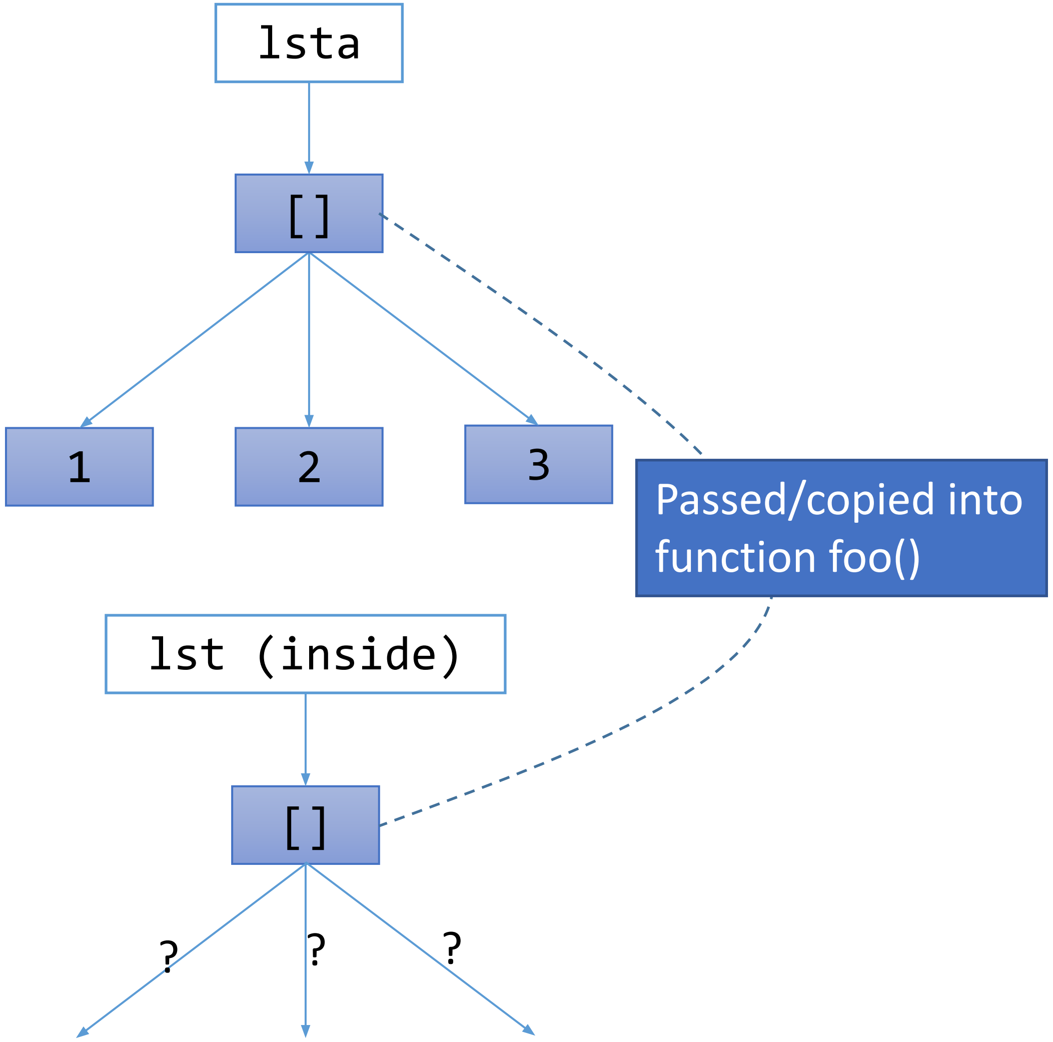


```
a = 4

def foo(x):
    x = x * 2
    print(x)

print(a)
foo(a)
print(a)
```

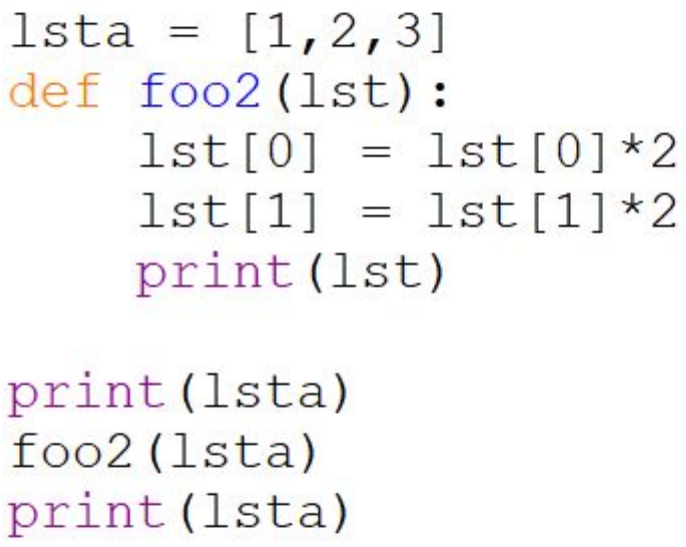
```
4
8
4
```



Copy what?



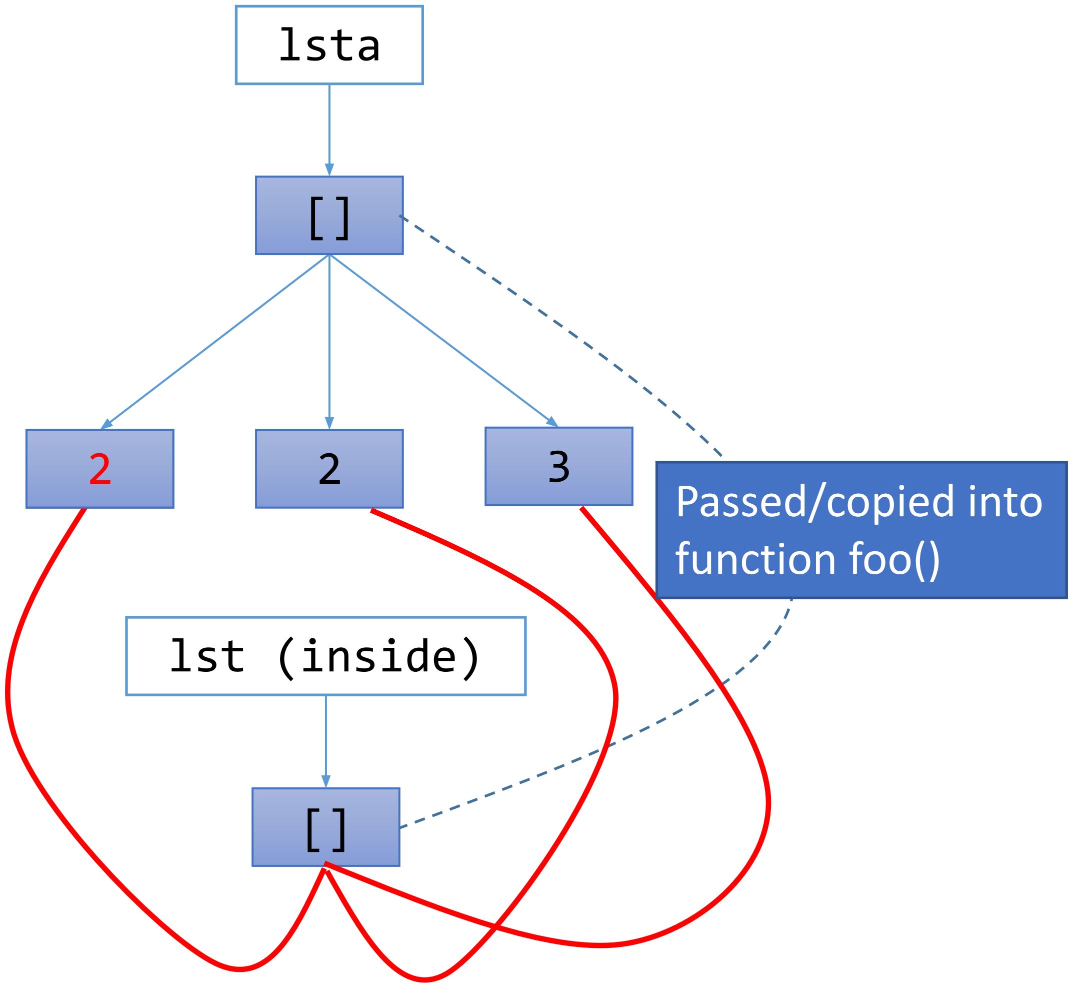
Passed/copied into function foo()



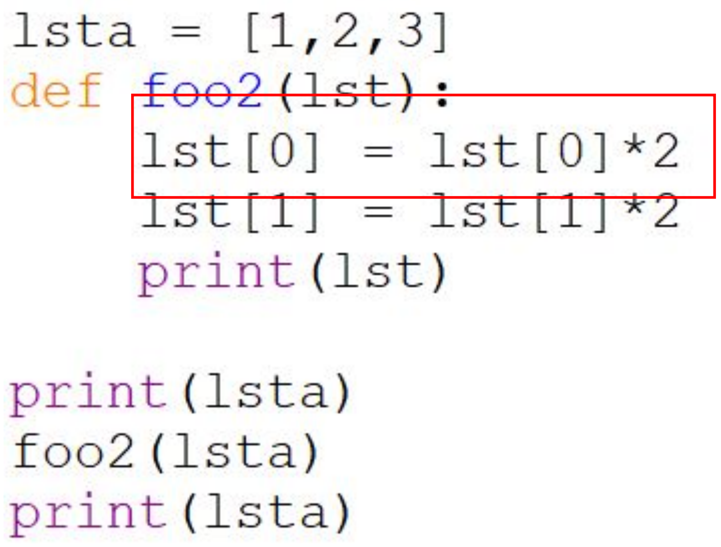
```
lsta = [1,2,3]
def foo2(lst):
    lst[0] = lst[0]*2
    lst[1] = lst[1]*2
    print(lst)

print(lsta)
foo2(lsta)
print(lsta)
```

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

Copy the REFERENCE (arrows)



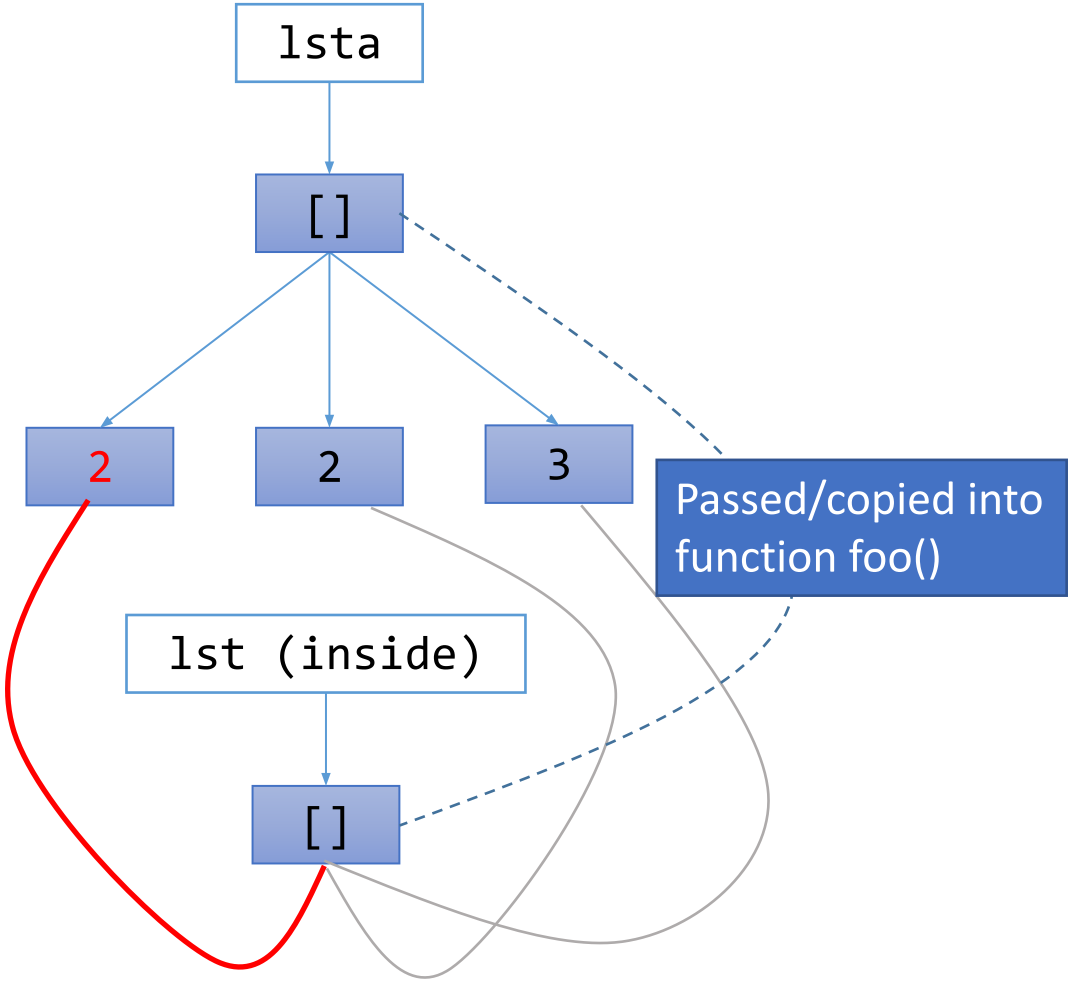
Passed/copied into function foo()



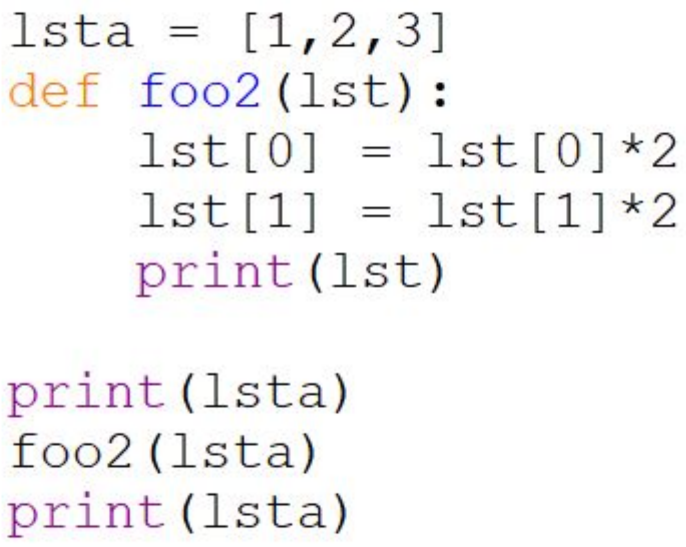
```
lsta = [1,2,3]
def foo2(lst):
    lst[0] = lst[0]*2
    lst[1] = lst[1]*2
    print(lst)

print(lsta)
foo2(lsta)
print(lsta)
```

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

Copy the REFERENCE (arrows)



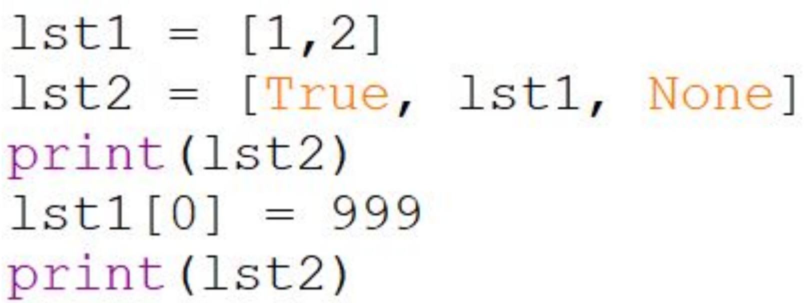
Passed/copied into function foo()



```
lsta = [1,2,3]
def foo2(lst):
    lst[0] = lst[0]*2
    lst[1] = lst[1]*2
    print(lst)

print(lsta)
foo2(lsta)
print(lsta)
```

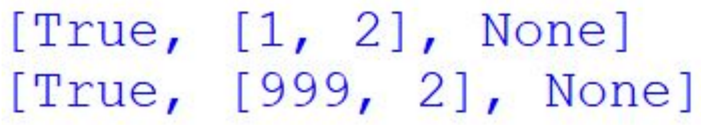
```
[1, 2, 3]
[2, 4, 3]
[2, 4, 3]
```

Same Idea

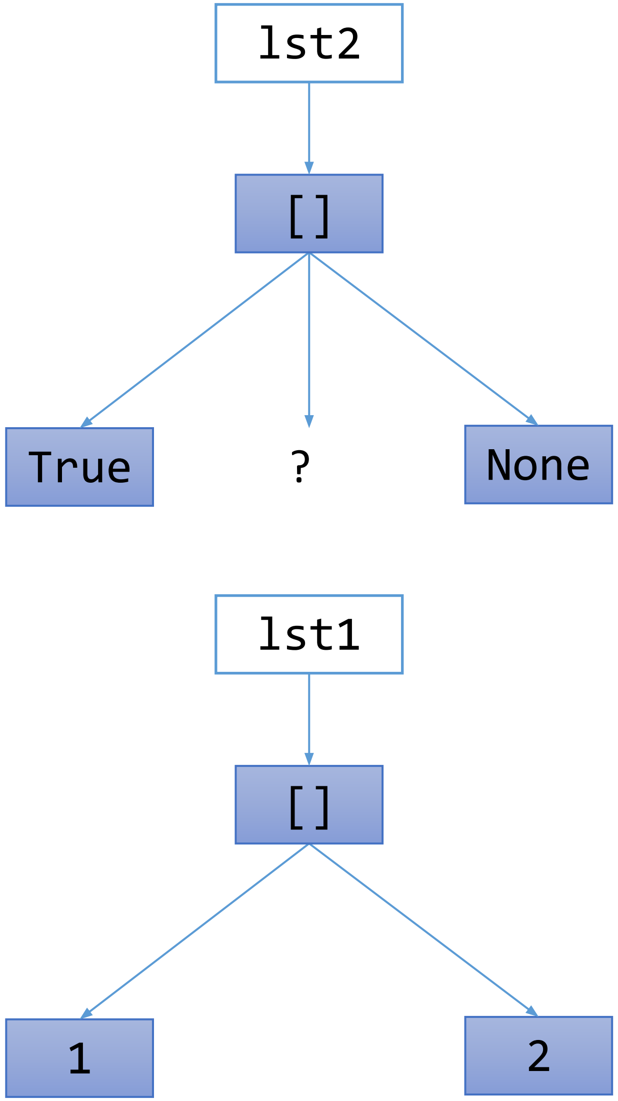


```
lst1 = [1,2]
lst2 = [True, lst1, None]
print(lst2)
lst1[0] = 999
print(lst2)
```

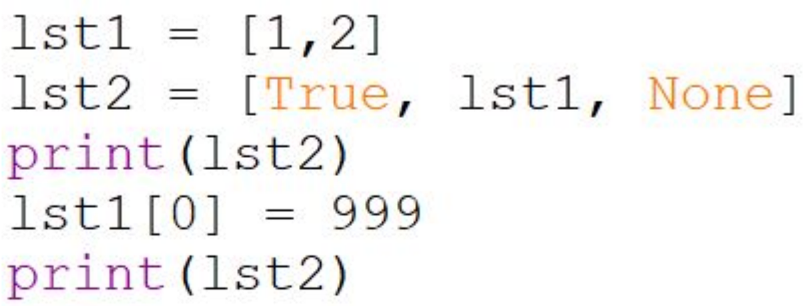
- Output



```
[True, [1, 2], None]
[True, [999, 2], None]
```

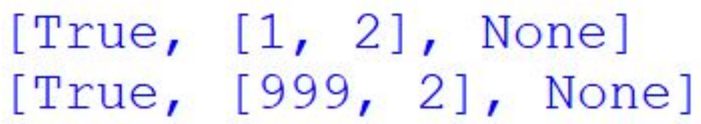


Same Idea

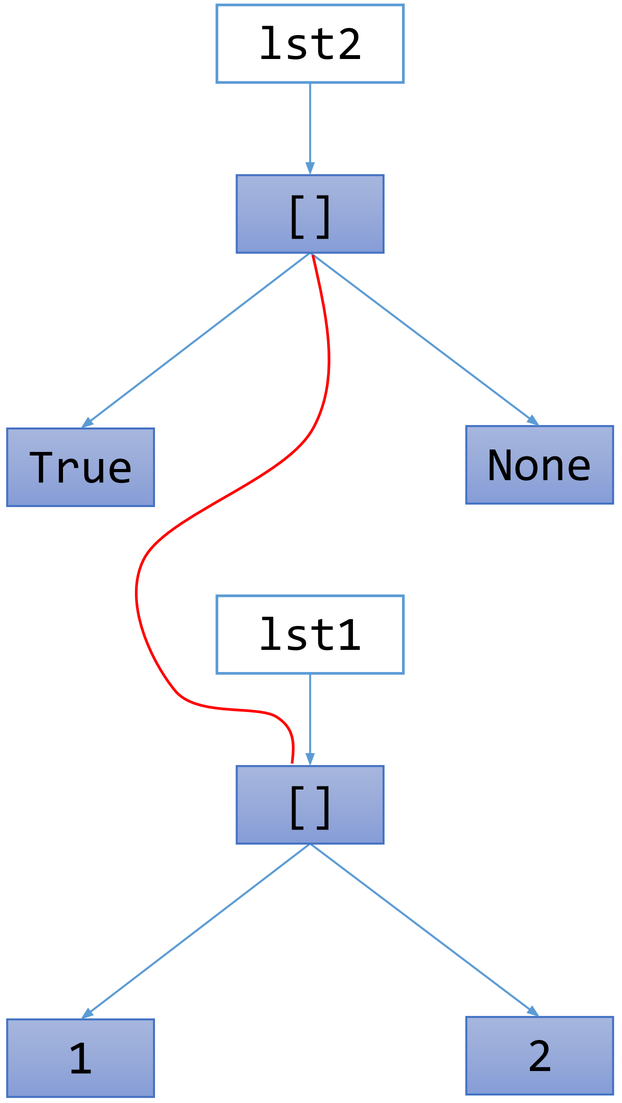


```
lst1 = [1,2]
lst2 = [True, lst1, None]
print(lst2)
lst1[0] = 999
print(lst2)
```

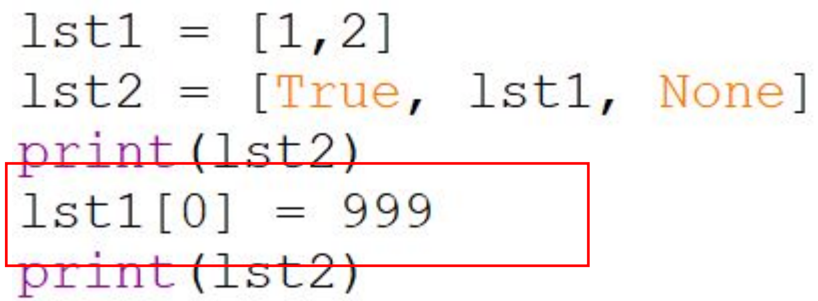
- Output



```
[True, [1, 2], None]
[True, [999, 2], None]
```

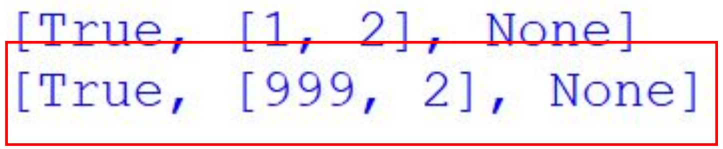


Same Idea

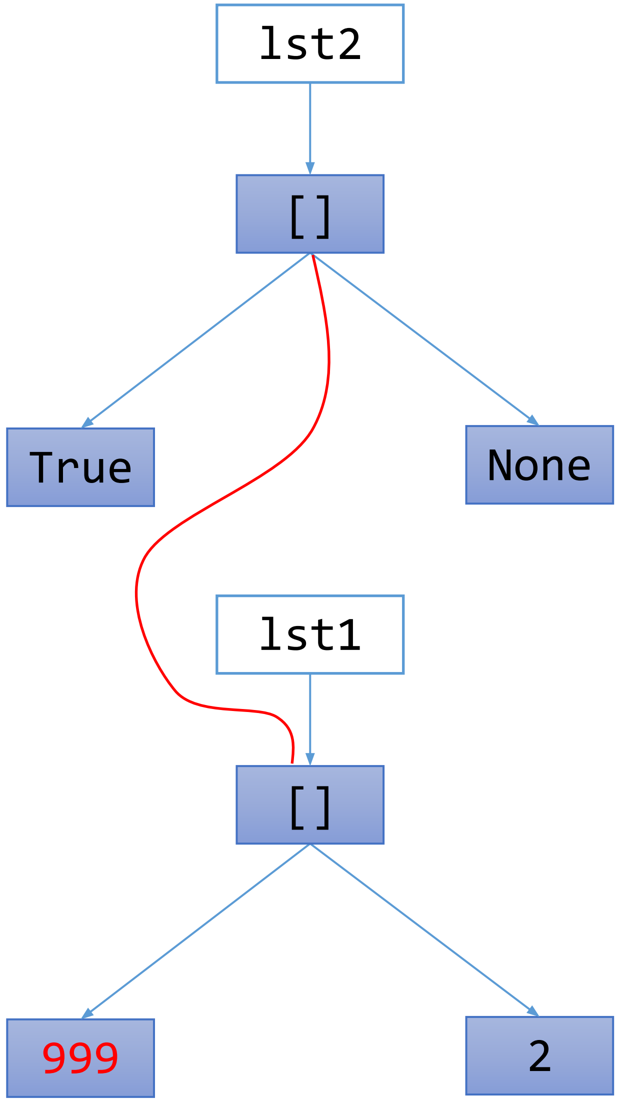


```
lst1 = [1,2]
lst2 = [True, lst1, None]
print(lst2)
lst1[0] = 999
print(lst2)
```

- Output

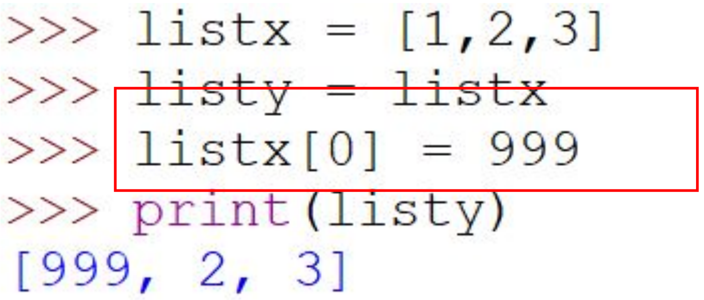


```
[True, [1, 2], None]
[True, [999, 2], None]
```



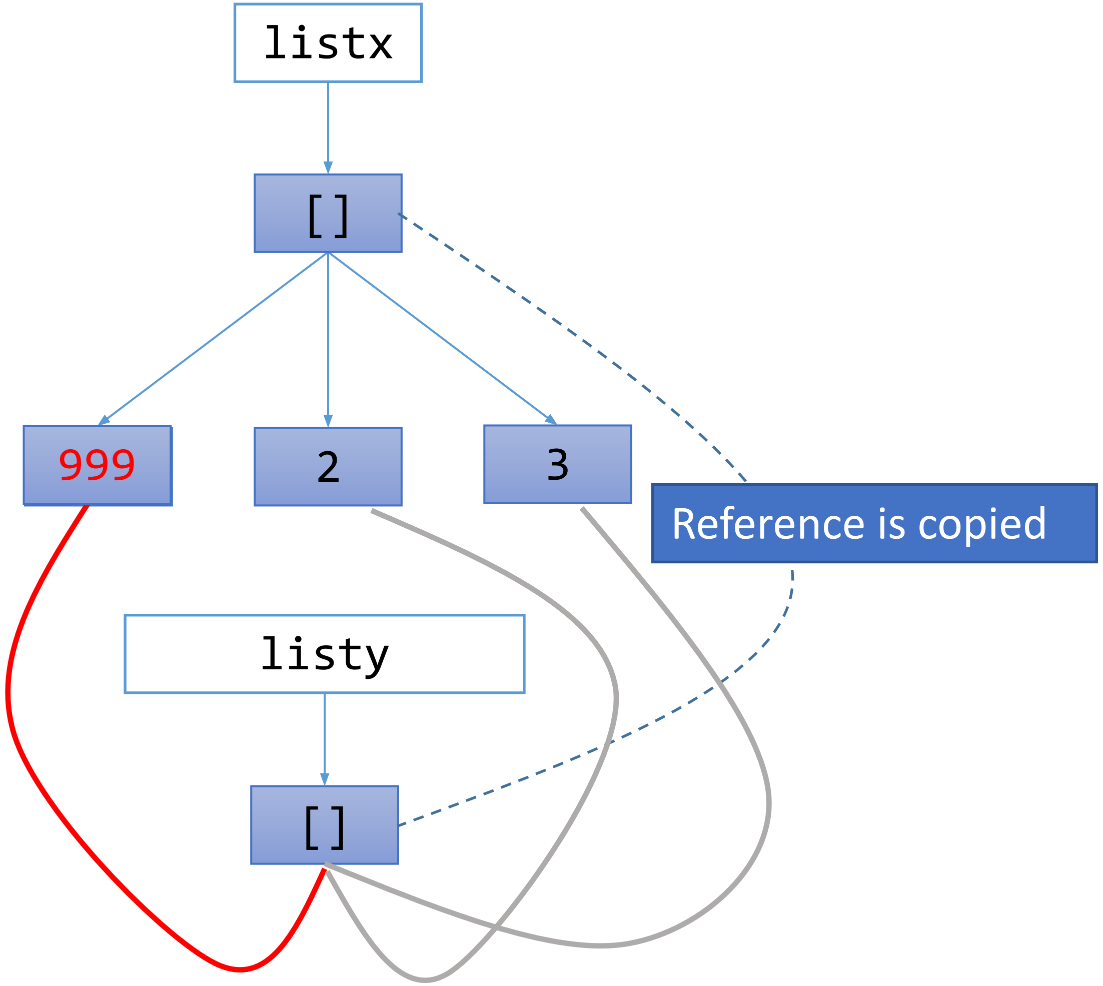
How do we AVOID this?

- For list



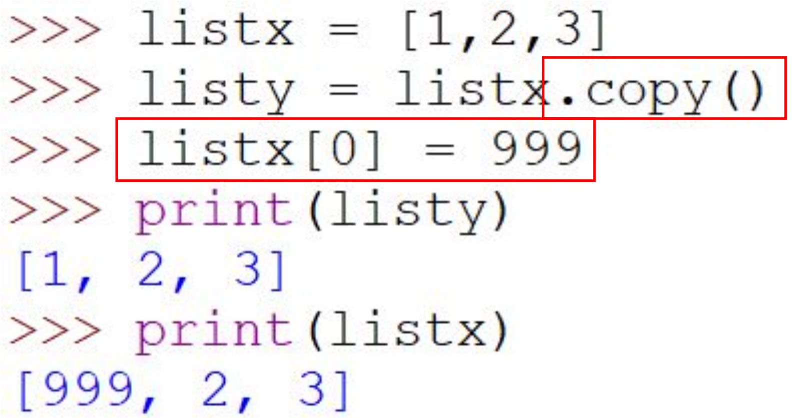
```
>>> listx = [1,2,3]
>>> listy = listx
>>> listx[0] = 999
>>> print(listy)
[999, 2, 3]
```

- We change `listx`
  - But `listy` is also changed?



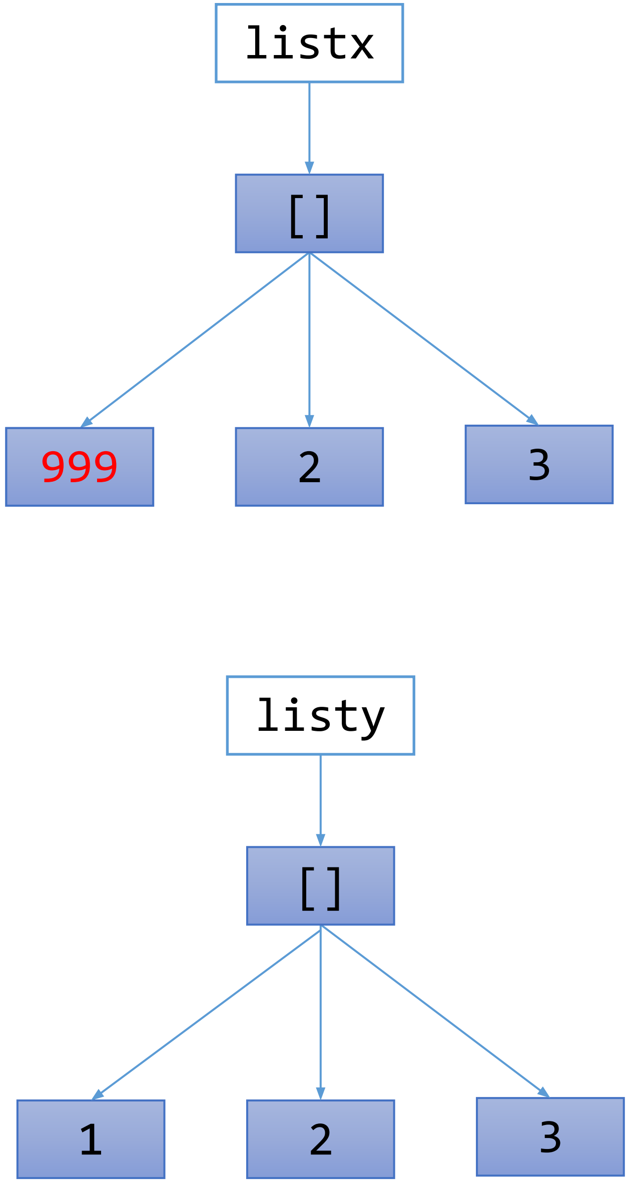
Reference is copied

Use the function copy()

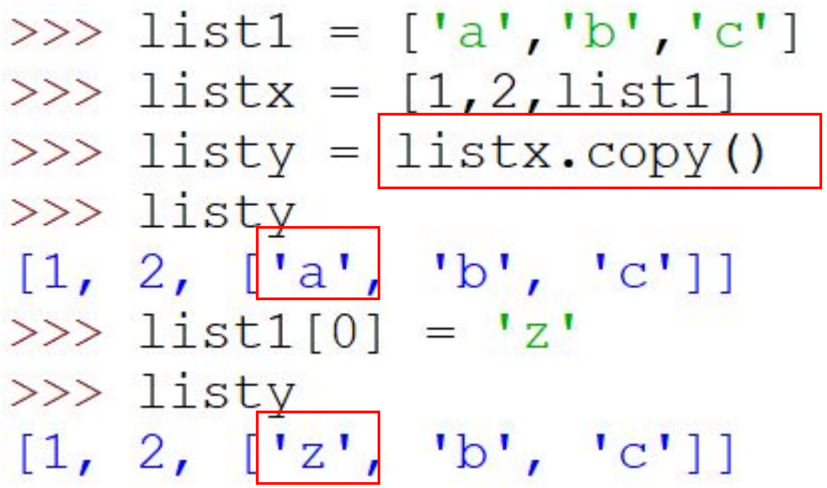


```
>>> listx = [1,2,3]
>>> listy = listx.copy()
>>> listx[0]  = 999
>>> print(listy)
[1, 2, 3]
>>> print(listx)
[999, 2, 3]
```

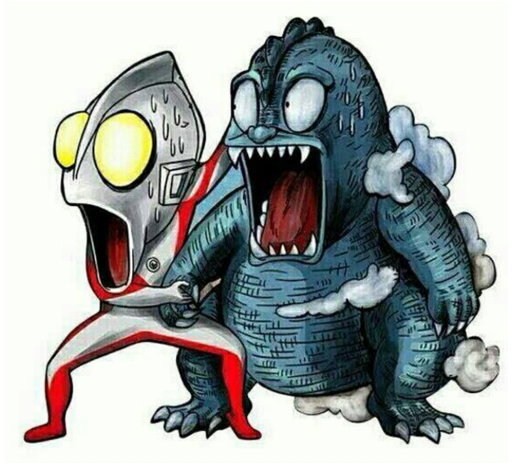
- “copy()” means to make a duplicate



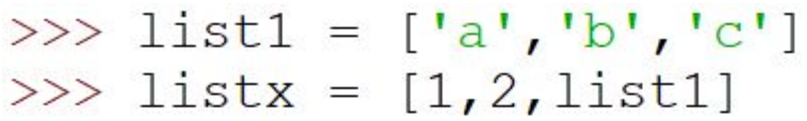
However, life is not easy



```
>>> list1 = ['a','b','c']
>>> listx = [1,2,list1]
>>> listy = listx.copy()
>>> listy
[1, 2, ['a', 'b', 'c']]
>>> list1[0] = 'z'
>>> listy
[1, 2, ['z', 'b', 'c']]
```

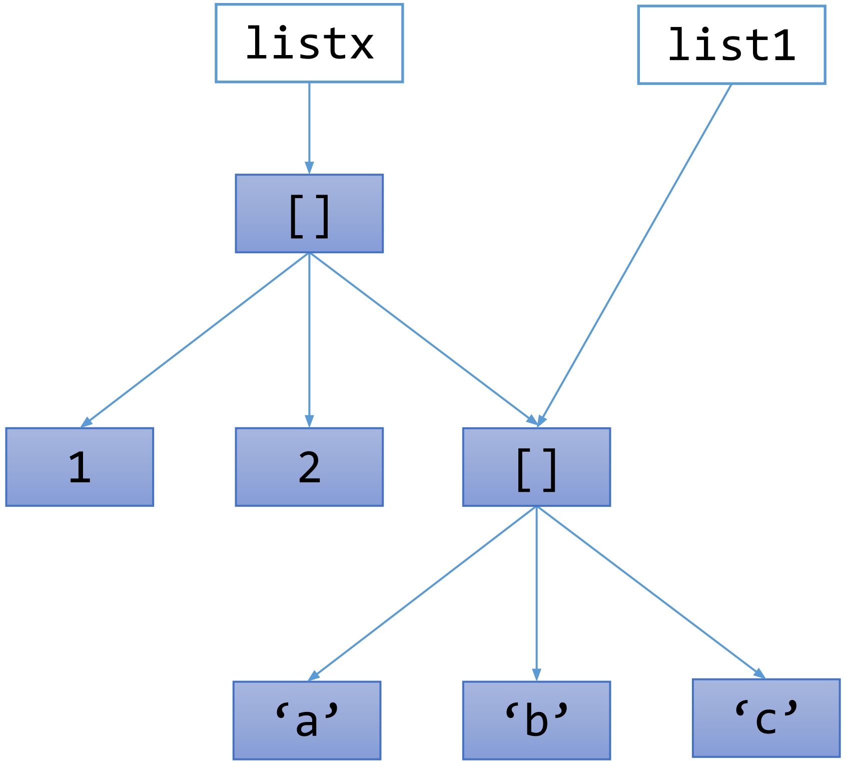


However, life is not easy

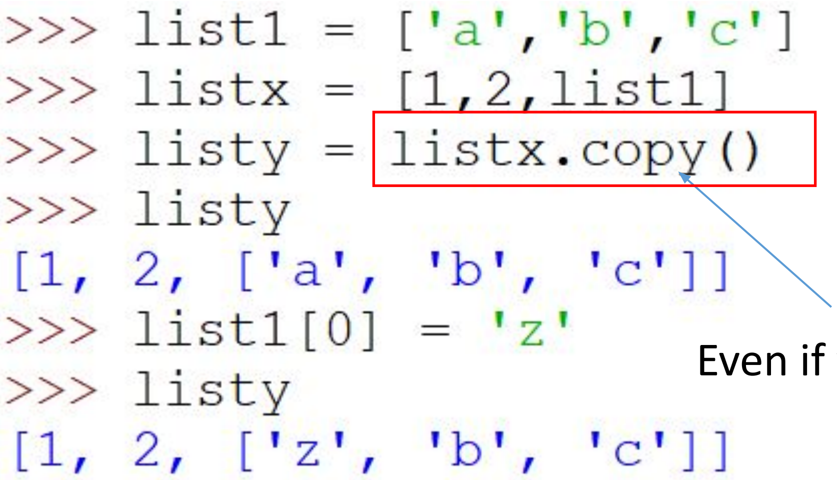


```
>>> list1 = ['a','b','c']
>>> listx = [1,2,list1]
```

Did not use “copy()”

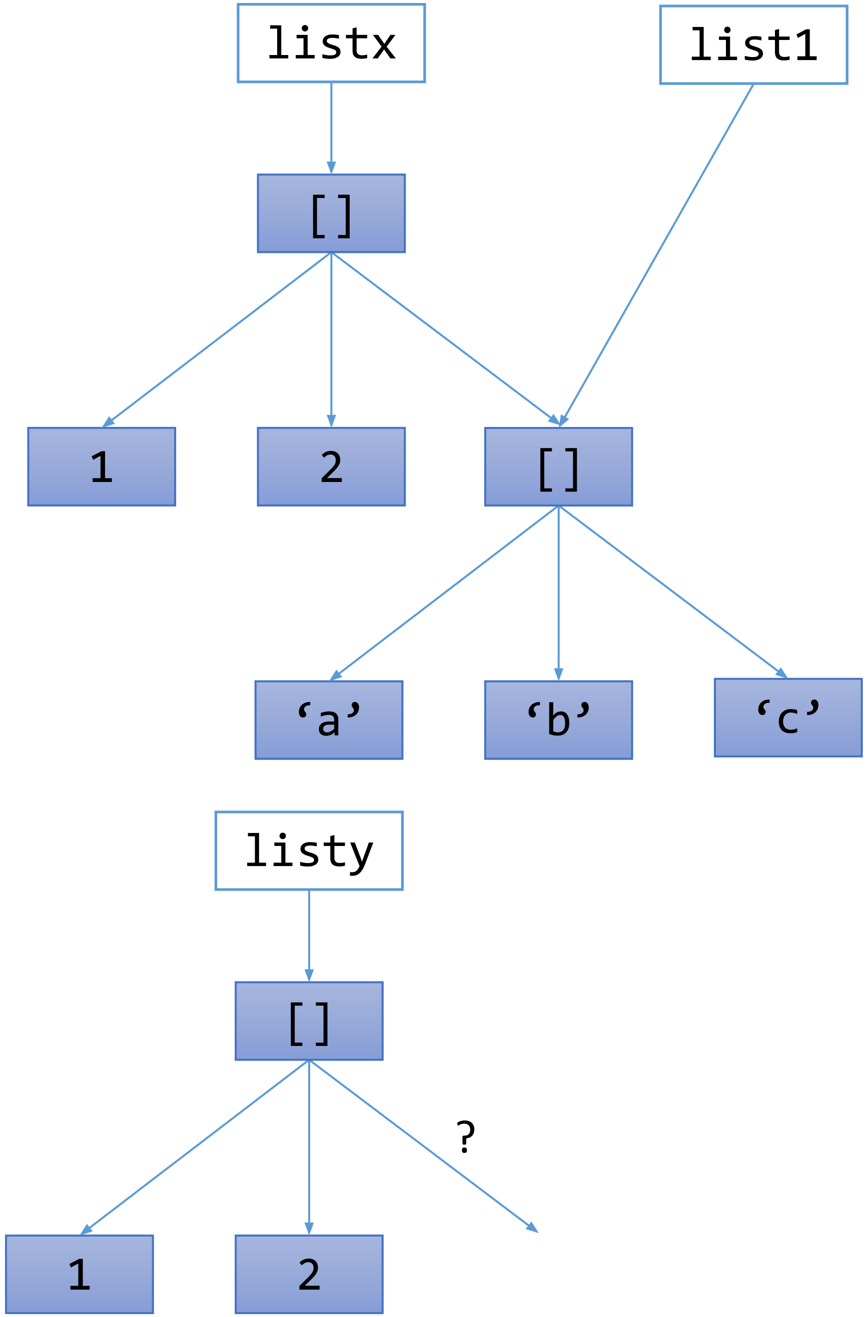


However, life is not easy

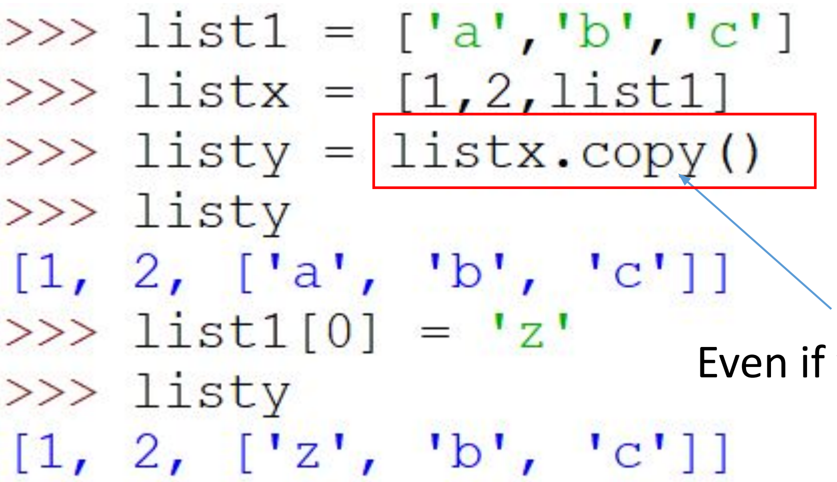


```
>>> list1 = ['a','b','c']
>>> listx = [1,2,list1]
>>> listy = listx.copy()
>>> listy
[1, 2, ['a', 'b', 'c']]
>>> list1[0] = 'z'
>>> listy
[1, 2, ['z', 'b', 'c']]
```

Even if you use “copy()”

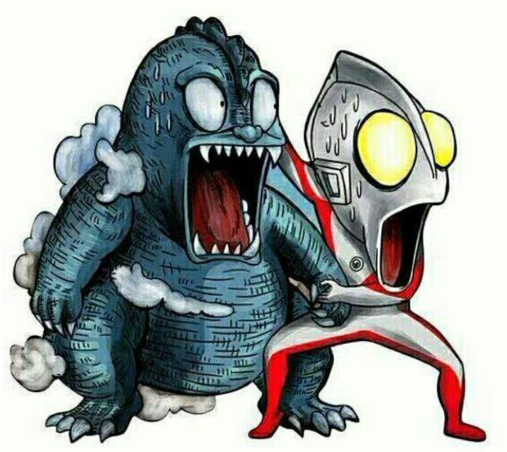


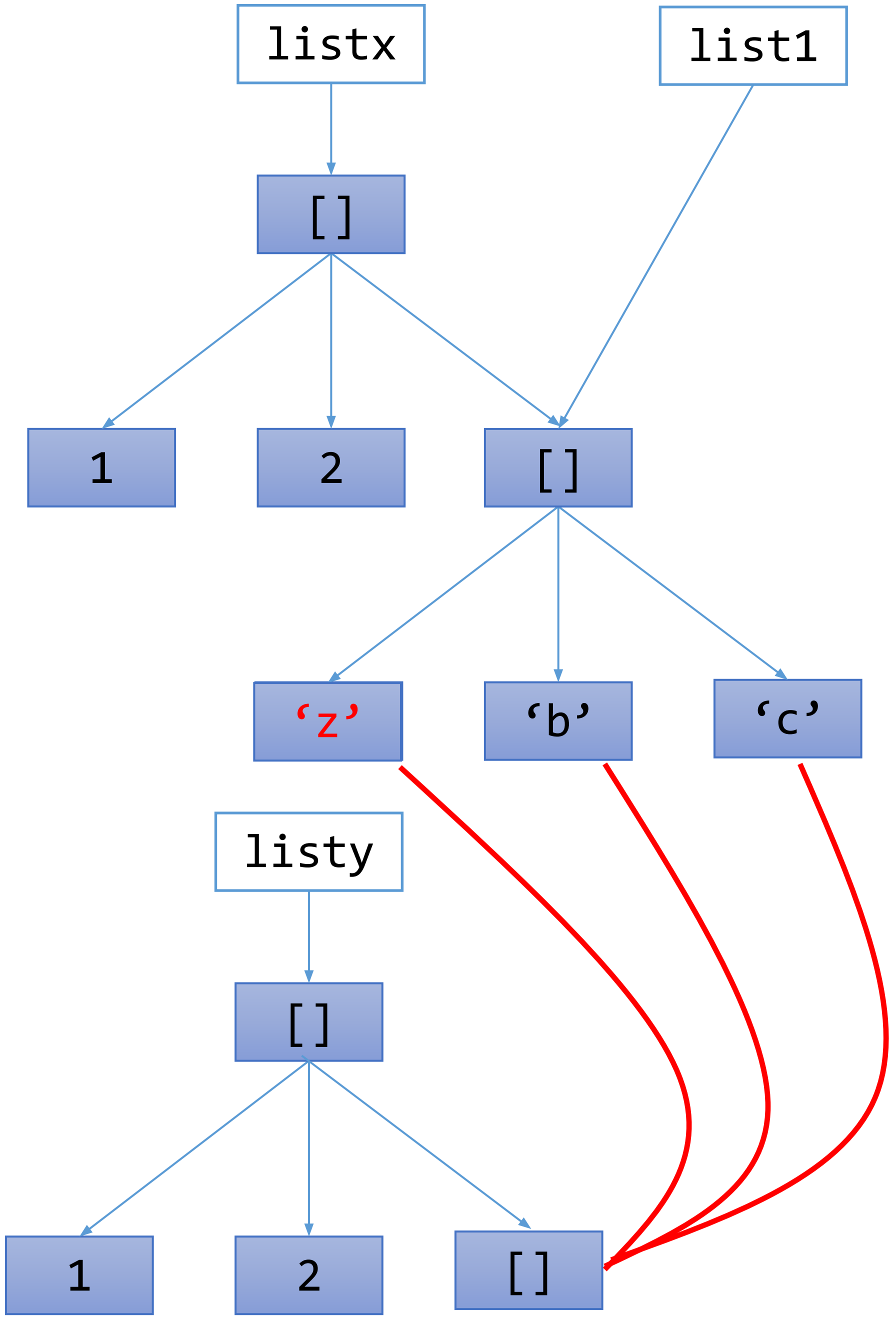
The Truth



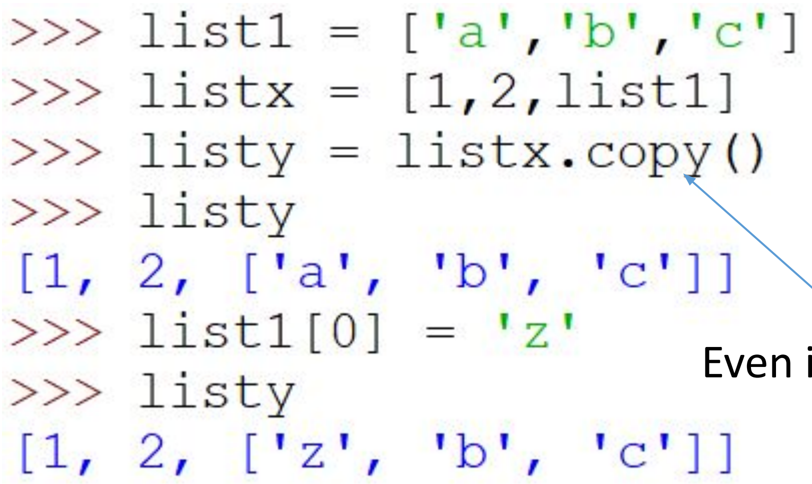
```
>>> list1 = ['a','b','c']
>>> listx = [1,2,list1]
>>> listy = listx.copy()
>>> listy
[1, 2, ['a', 'b', 'c']]
>>> list1[0] = 'z'
>>> listy
[1, 2, ['z', 'b', 'c']]
```

Even if you use “copy()”





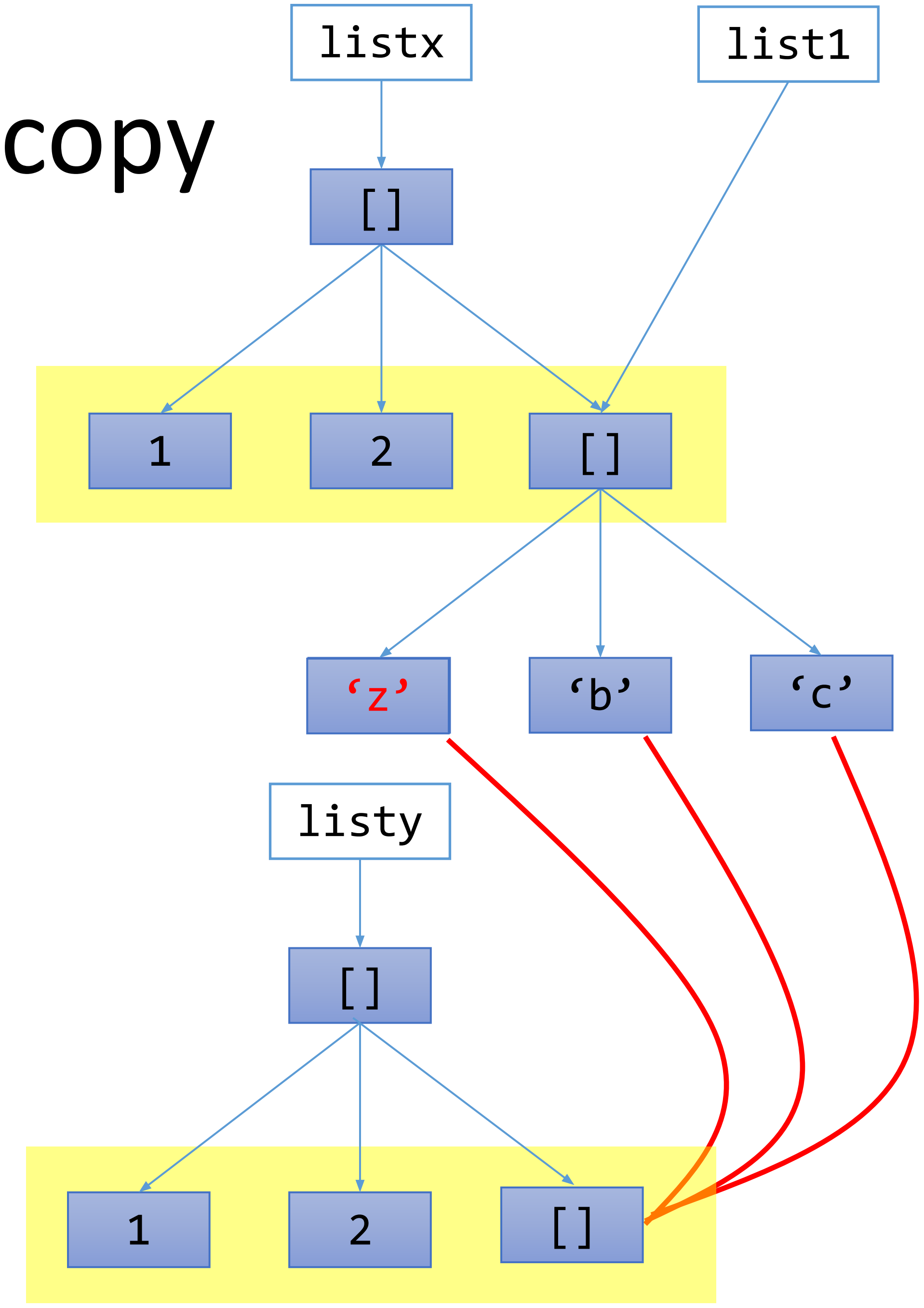
“copy()” is only a SHALLOW copy



```
>>> list1 = ['a','b','c']
>>> listx = [1,2,list1]
>>> listy = listx.copy()
>>> listy
[1, 2, ['a', 'b', 'c']]
>>> list1[0] = 'z'
>>> listy
[1, 2, ['z', 'b', 'c']]
```

Even if you use “copy()”

- Only the first layer is **duplicated**

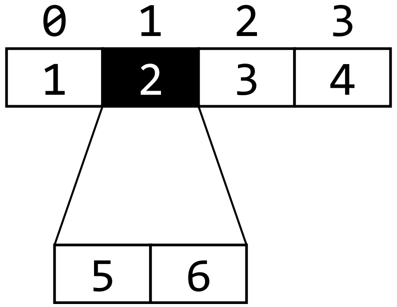


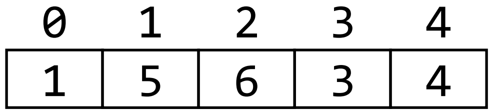
Pointers

- The list block diagram is essential to understanding a very important concept in computer programming
  - Namely, pointers (memory address) e.g. in C
- In Python, this *hideous* concept is “well-encapsulated” from beginners
- To advance in programming, learning about pointers is unavoidable
- Also, this gives us more motivation to use tuples rather than lists
  - Because tuples do not have this complication

List Mutation with Slicing

- To modify an element in a list at index i
  - lst[i] = val
- Python also allows you to insert a list (iterable) into another list
  - This is done via modified slicing
  - Let lst = [1, 2, 3, 4]
  - lst[1:2] = [5, 6]   lst = [1, 5, 6, 3, 4]





Exercise: Course Schedule

Tuples and Lists

Problem

- You are provided with an implementation for NUS courses:

```
def make_course(code, units):
  return (code, units)
def make_units(lec, tut, lab, hw, prep):
  return (lec, tut, lab, hw, prep)
def get_course_code(course):
  return course[0]
def get_course_units(course):
  return course[1][0] + course[1][1] + course[1][2] + \
         course[1][3] + course[1][4]
```

Problem

- Each course has a code and an associated number of credit units
  - For instance, the credit units for CS1010E are 2-1-1-3-3
- Your job is to write a schedule object to represent the courses taken by a student
- In your code, you should respect abstraction barriers
  - To get the course code, you cannot use `course[0]` but must call the function `get_course_code(course)`

Tasks

1. Write a function `make_empty_schedule()` that returns an empty schedule of courses
2. Write a function `add_course(course, schedule)` that returns a new schedule with the added course
3. Write a function `total_scheduled_units(schedule)` that returns the total number of units of all courses in the schedule
4. Write a function `drop_course(course, schedule)` that returns a new schedule without the specified course
5. Write a function `valid_schedule(schedule, max_units)` that returns a new schedule with the total number of units less than or equal to `max_units` by removing courses from the specified schedule

Challenge: Extra Tasks

- **Qn2:** If `course` is already in the schedule, return the schedule as is
- **Qn3:** Write an iterative and a recursive version of the function
- **Qn4:** Write an iterative and a recursive version of the function
- **Qn4:** If there are duplicate courses to be removed, drop ALL
- **Qn5:** Return a schedule with the ***maximum*** number of units the student can take by removing courses from the schedule