# Sequence

tuple, list, iterables

# Tuple

# Quick Tuple Exercise

**Code**

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

**Output**

# Quick Tuple Exercise

**Code**

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

**Output**

```
(10, 11, 12, 13)

(“CS”, 1010)

(10, 11, 12, 13, “CS”, 1010)
6
```

# Quick Tuple Exercise

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

**Code**

```
print(11 in tup_a)
print(14 in tup_b)
print(“C” in tup_c)
print(tup_b[1])
tup_d = tup_b[0]*4
print(tup_d)
print(tup_b[1] * 4)
```

**Output**

# 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` is NOT a tuple!!!!

# Quick Tuple Exercise

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

**Code**

```
tup_e = tup_d[1:]
print(tup_e)
tup_f = tup_d[::-1]
print(tup_f)
tup_g = tup_d[1:-1:2]
print(tup_g)
tup_h = tup_d[-1:6:-2]
print(tup_h)
```

**Output**

# Quick Tuple Exercise

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

| **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’` |

`tup_e` is NOT a tuple!!!!

# Quick Tuple Exercise

**Code**

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

**Output**

# 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)` |

`tup_i` is NOT a tuple!!!!

# Quick Tuple Exercise

```
tup_a = (10, 11, 12, 13)
tup_b = (“CS”, 1010)
tup_c = tup_a + tup_b
tup_d = ‘CSCSCSCS’
tup_e = tup_d[1:]
```

**Code**

```
print(min(tup_a))
print(max(tup_a))
print(min(tup_c))
print(max(tup_c))
print(min(tup_e))
print(max(tup_e))
```

**Output**

# Quick Tuple Exercise

```
tup_a = (10, 11, 12, 13)
tup_b = (“CS”, 1010)
tup_c = tup_a + tup_b
tup_d = ‘CSCSCSCS’
tup_e = tup_d[1:]
```

| **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’` |

# Quick Tuple Exercise

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

**Code**

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

**Output**

# Quick Tuple Exercise

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

**Code**

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

**Output**

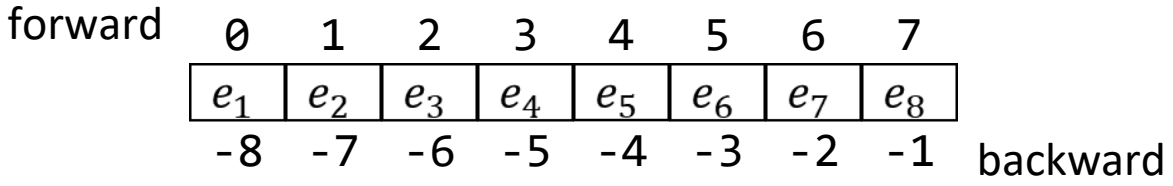
```
‘CS’
1010
```

# Tuple

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

# Tuple Access

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




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

# Tuple Access Exercises

- Given the following tuple, 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 are the lengths of each of the tuple above?
- Which of the following will return `True`?
  - `4 in tup1`
  - `4 in tup2`
  - `4 in tup3`

# List

# Quick List Exercise

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

**Code**

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

**Output**

# Quick List Exercise

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

**Code**

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

**Output**

```
[“CS”, 1010]

[“E”,(“is”, “easy”)]

[“CS”, 1010, “E”,
  (“is”, “easy”)]
```

# Quick List Exercise

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

**Code**

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

**Output**

# Quick List Exercise

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

**Code**

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

**Output**

TypeError

[“CS”, 2030]

# Quick List Exercise

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

**Code**

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

**Output**

# Quick List Exercise

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

**Code**

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

**Output**

```
[“CS”, 1010, “easy”]

[“CS”, 1010, “easy”, “e”, “a”,
 “s”, “y”]
```

# Quick List Exercise

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

**Code**

```
cpy_b = lst_b[:]
print(cpy_b)
cpy_b[1] = “is hard”
print(cpy_b)
print(lst_b)
```

**Output**

# Quick List Exercise

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

**Code**

```
cpy_b = lst_b[:]
print(cpy_b)
cpy_b[1] = “is hard”
print(cpy_b)
print(lst_b)
```

**Output**

```
[“E”, (“is”, “easy”)]

[“E”, “is hard”]
[“E”, (“is”, “easy”)]
```

# List

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

# List

Meaning changing the original list

- Mutation; given a list `lst`
  - `lst.append(x)` modifies list by adding an element x (*no return*)
  - `lst.extend(x)` modifies list by adding another list x (*no return*)
  - `lst.reverse()` modifies `lst` by reversing it (*no return*)
  - `lst.insert(i,x)` insert 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)` modifies `lst` by removing first occurrence of x
  - `lst.clear()` empties the list `lst`
- Except
  - `lst.copy()` returns a shallow copy of `lst`

# List

- Mutation; given a list `lst`
  - `lst.append(x)` modifies list by adding an element x (*no return*)
  - `lst.extend(x)` modifies list by adding another list x (*no return*)
- Difference?

# List

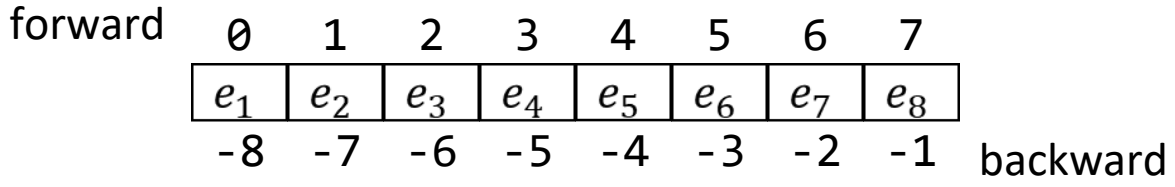
- Mutation; given a list `lst`
  - `lst.append(x)` modifies list by adding an element x (*no return*)
  - `lst.extend(x)` modifies list by adding another list x (*no return*)
- Difference?

```
>>> 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 index for list of size n
  - Forward index starts from 0, ends at n-1
  - Backward index starts from -1, ends at –n




- 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 of Loops?

- 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 of Loops?

- 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 of Loops?

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

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

# How about B?

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

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)

# Problem Example

- Given a list of numbers, to find the mean

```
>>> 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)

# Problem Example

- Given a list of numbers, to find the mean

```
>>> 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)
```

# Problem Example (Try it yourself)

```
>>> 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 to code `checkAllOddNum()`?
  - 10 min

# Mutable Objects in Python

# Try this

- For primitive data type

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

- For list

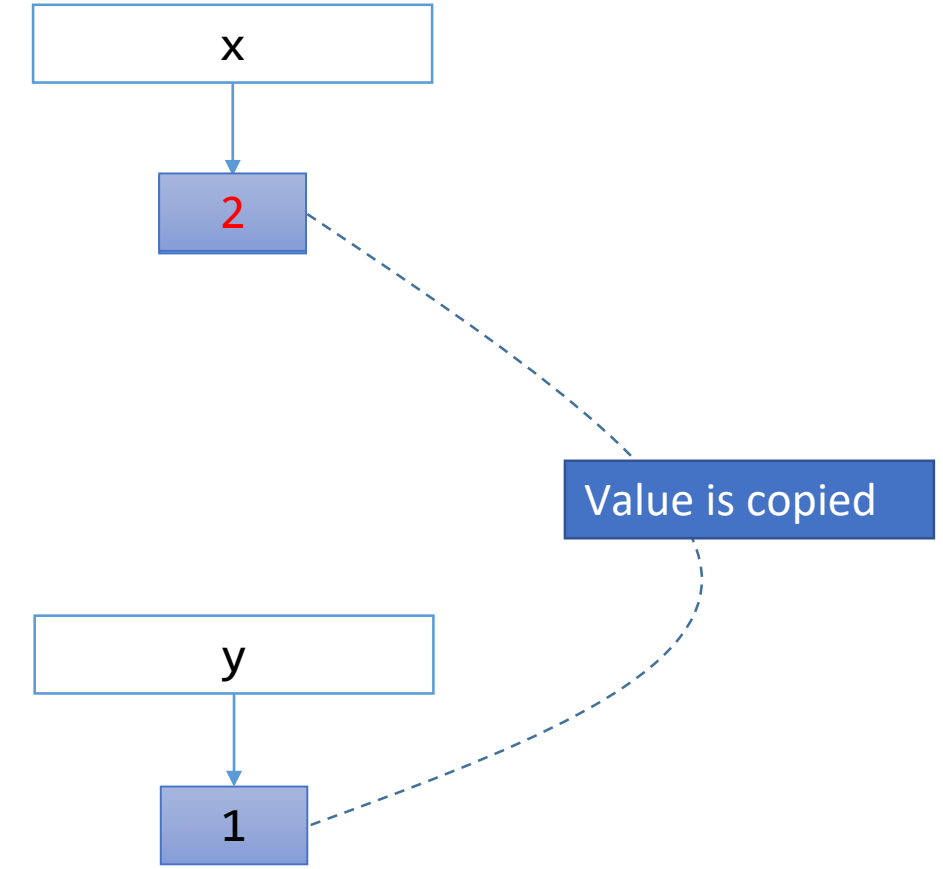
```
>>> listx = [1,2,3]
>>> listy = listx
>>> listx[0] = 999
>>> print(listy)
[999, 2, 3]
```

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

# For Primitive Data

- For primitive data type

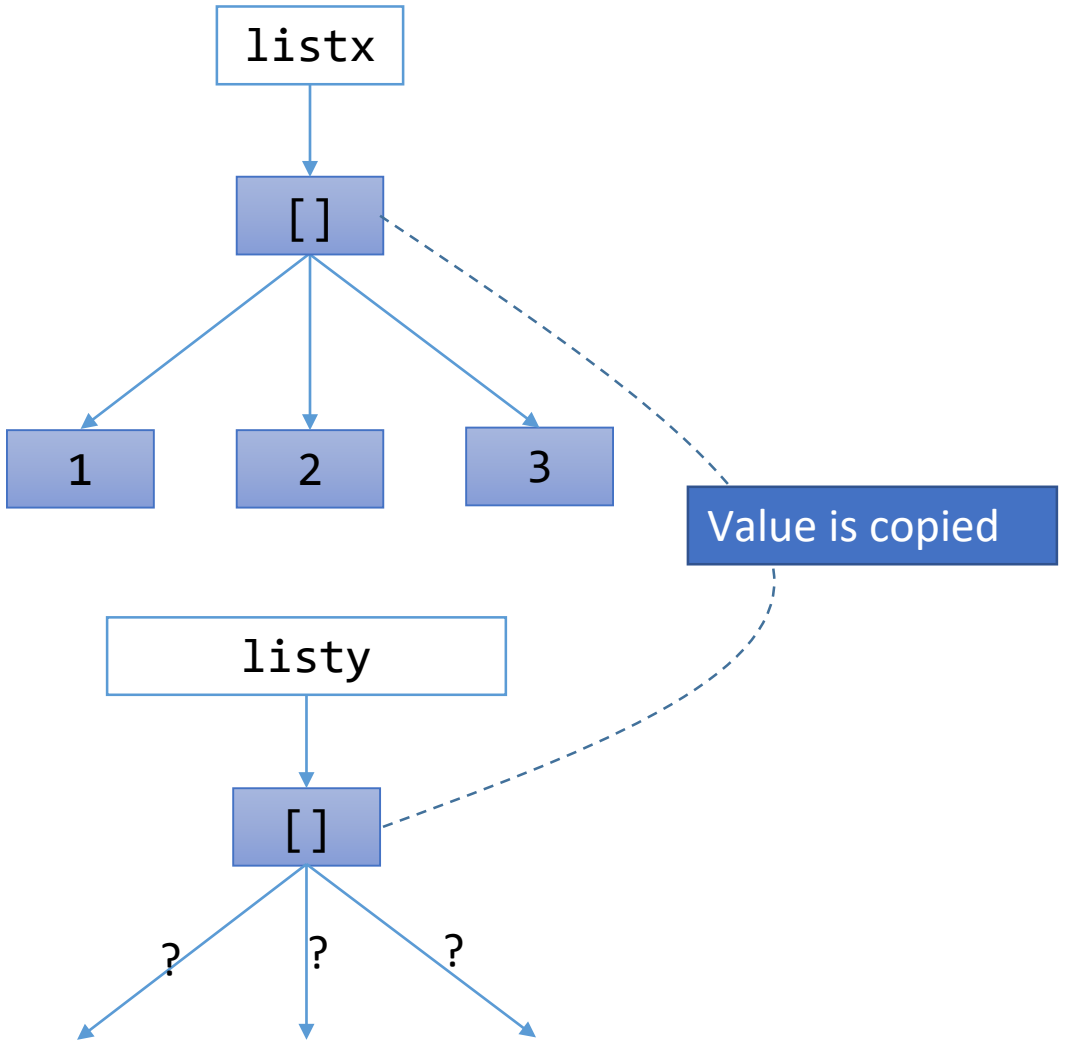
```
>>> x = 1
>>> y = x
>>> x = 2
>>> print(y)
1
```



# But for list

- For list

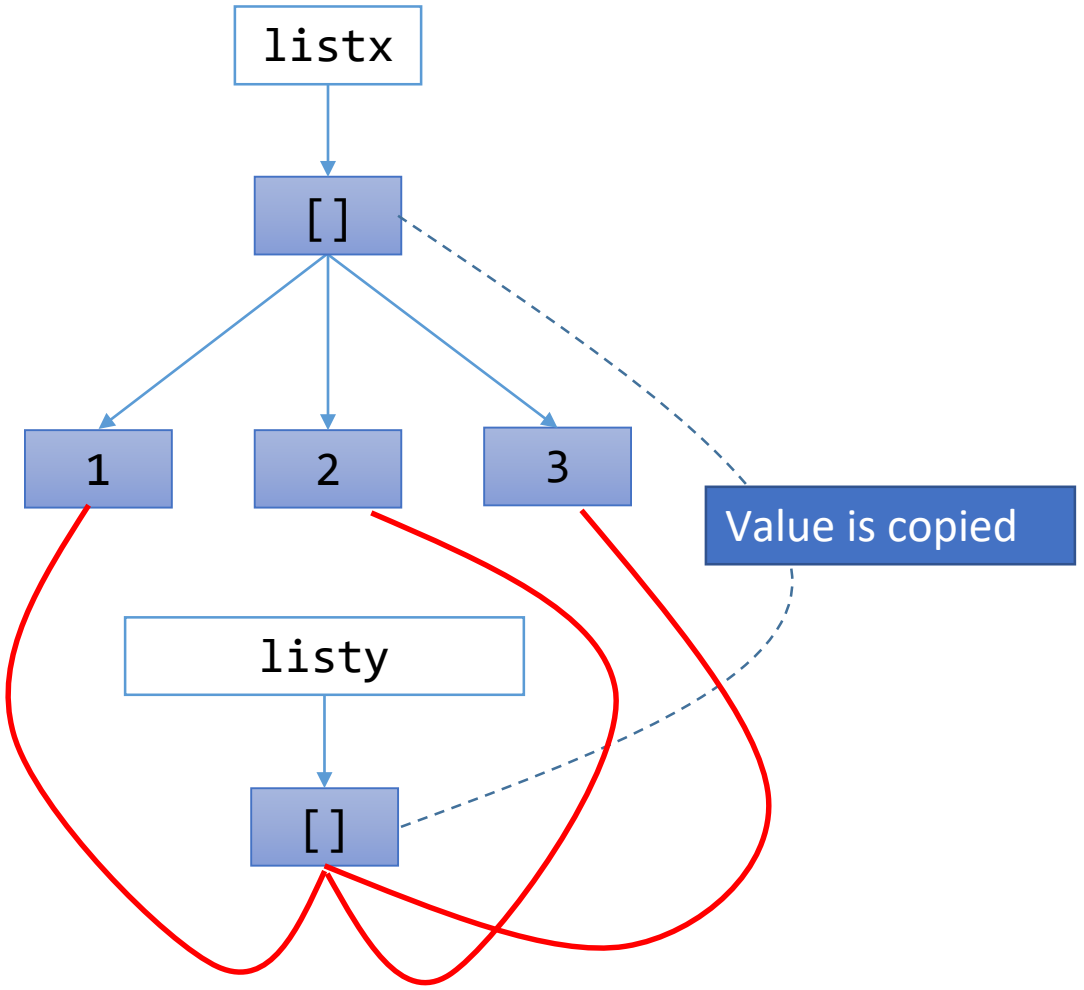
```
>>> listx = [1,2,3]
>>> listy = listx
```



# But for list

- For list

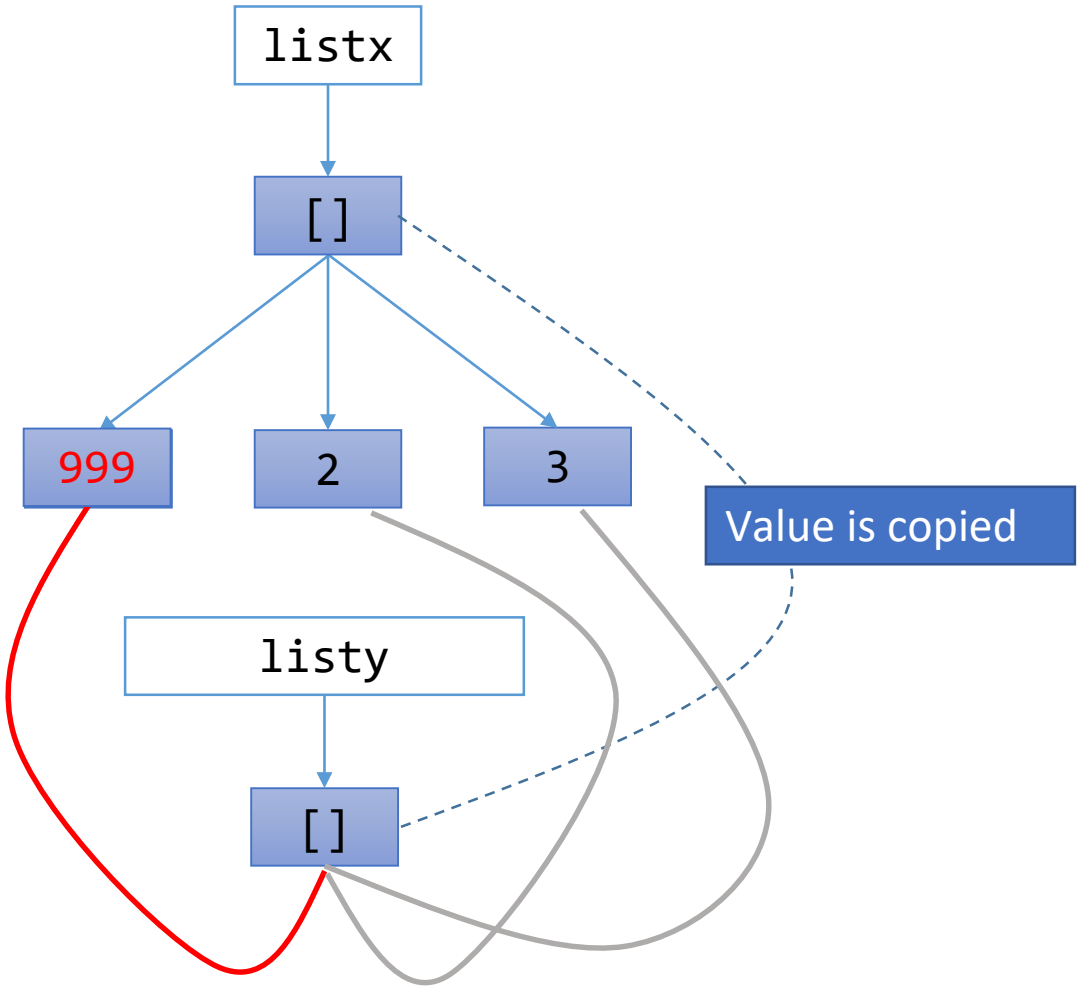
```
>>> listx = [1,2,3]
>>> listy = listx
```



# But for list

- For list

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



# Try this

```
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)
```

# Try this

```
a = 4

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

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

```
4
8
4
```

```
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]
```

Note the difference

# Try this

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

```
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]
```

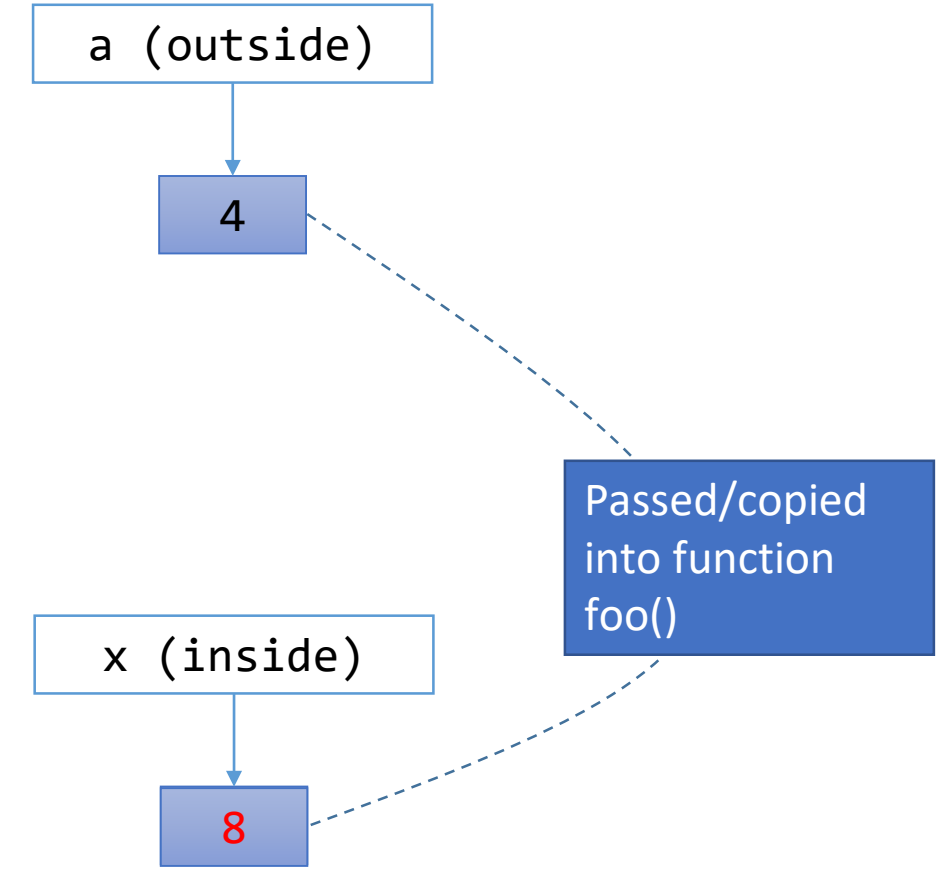
# By Block Diagram

```
a = 4

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

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

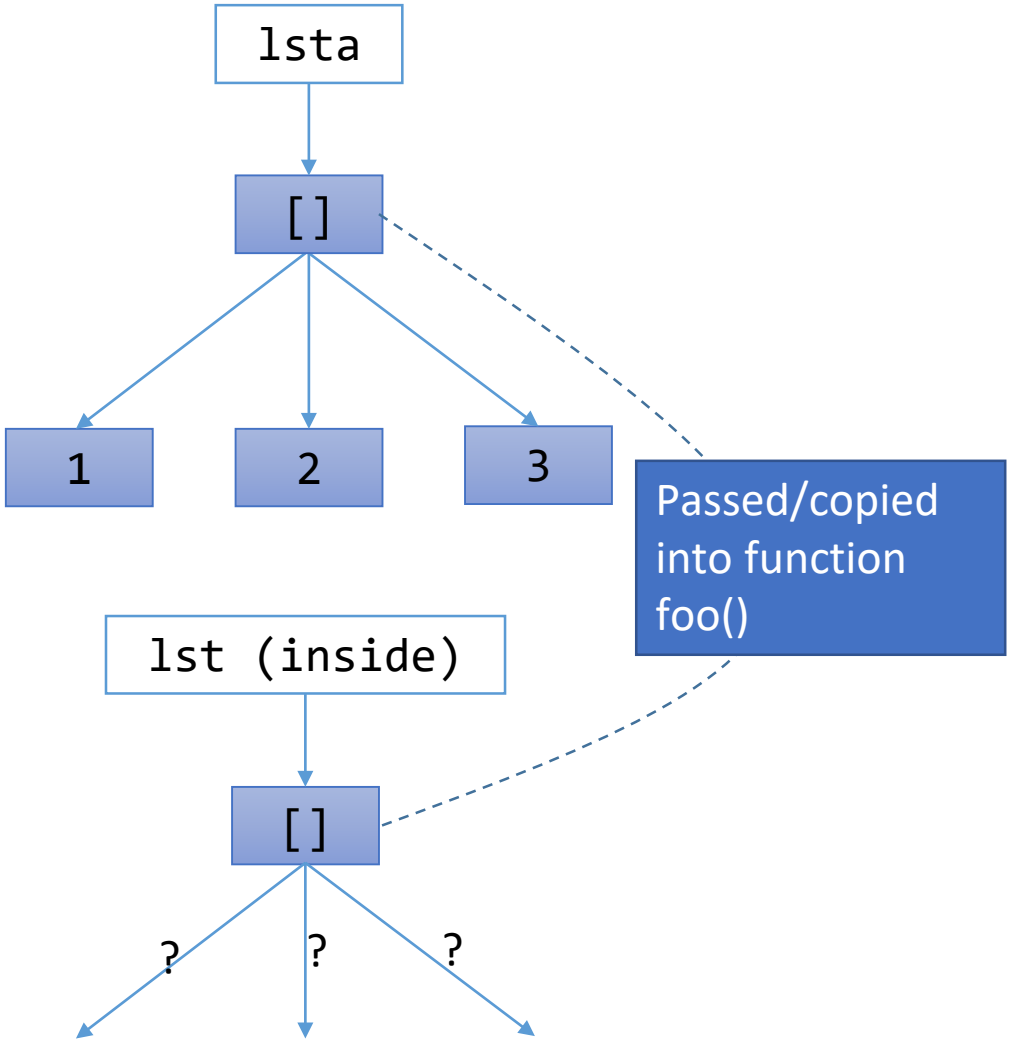
4

8

4



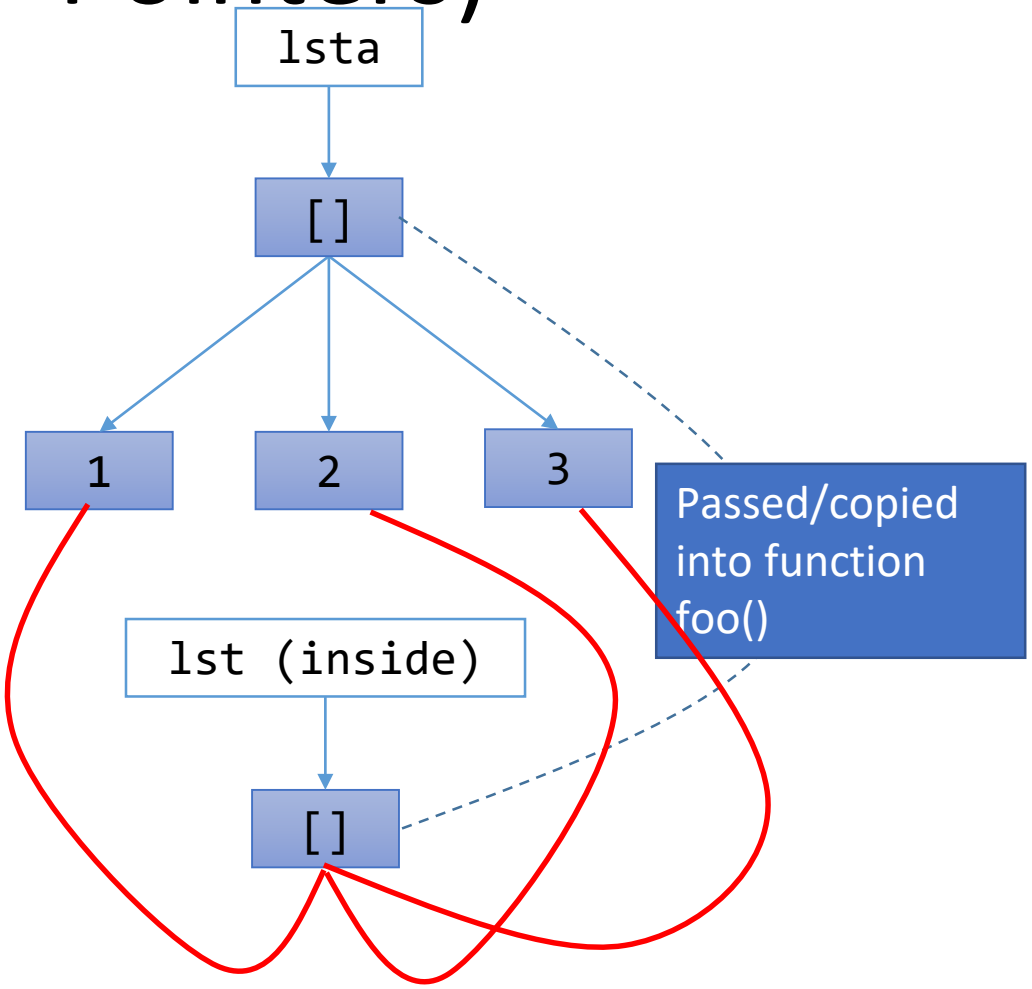
# Copy what?




```
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]
```

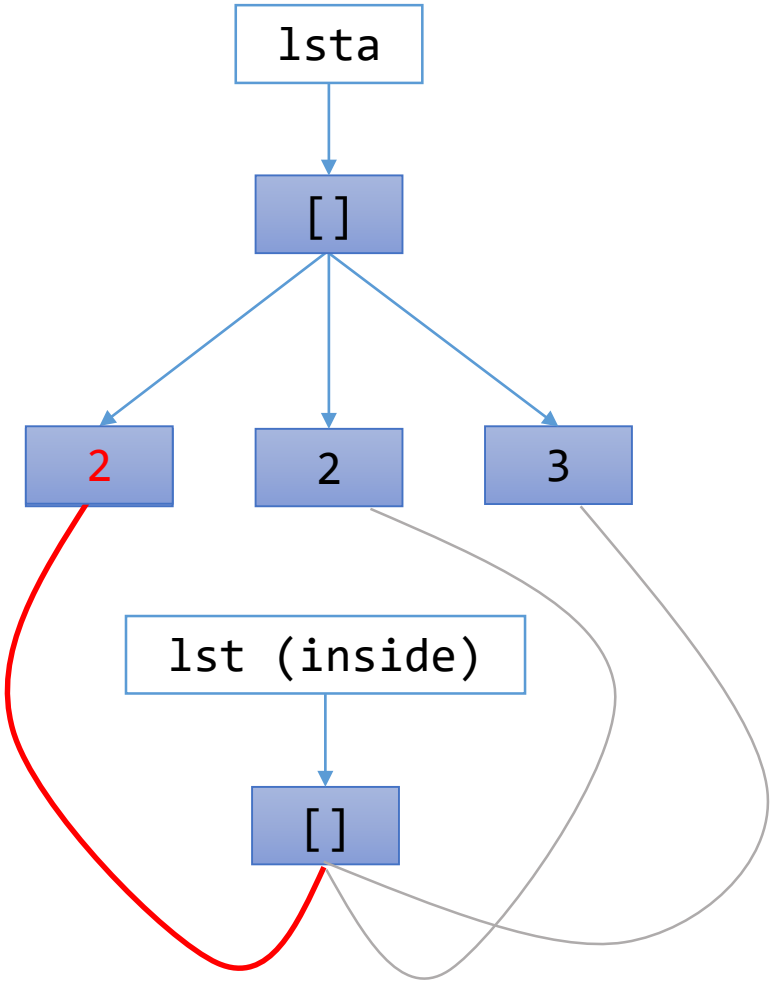
# Copy the ARROWS!!! (Formal name: Pointers)




```
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?



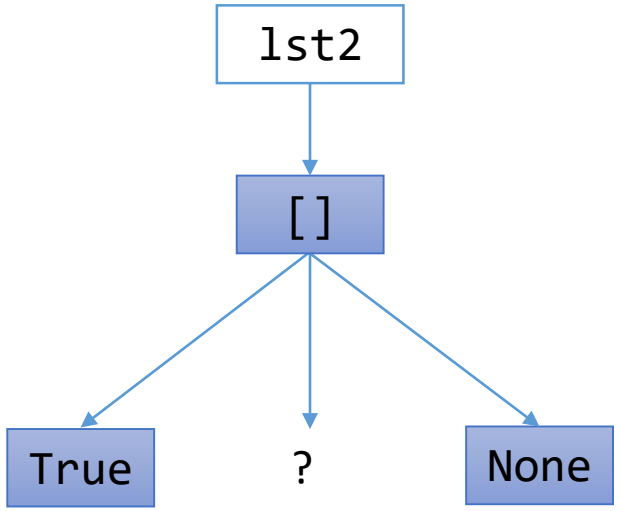

```
lsta = [1,2,3]
def foo2(lst):
    lst[0] = lst[0]*2
    lst[1] = lst[1]*2
    print(lst)

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

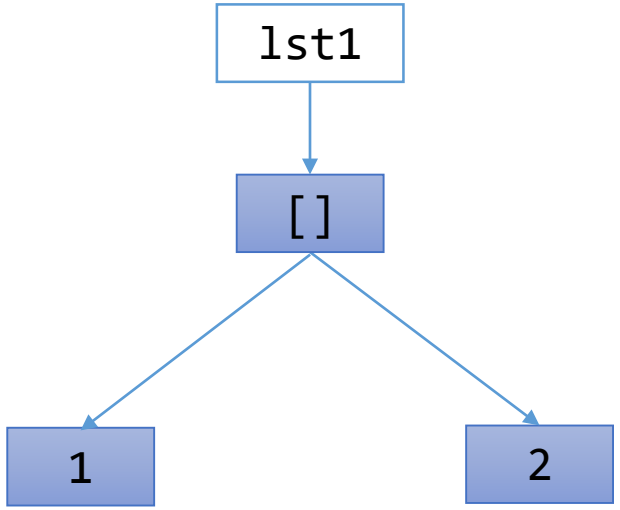
```
[1, 2, 3]
[2, 4, 3]
[2, 4, 3]
```

# Same Idea

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

- Output



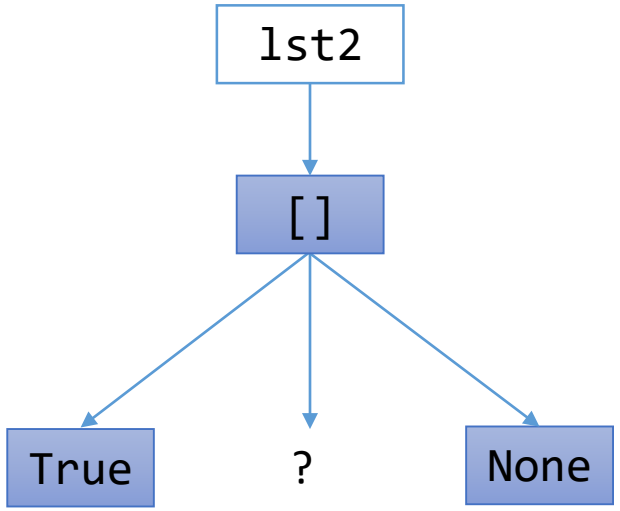



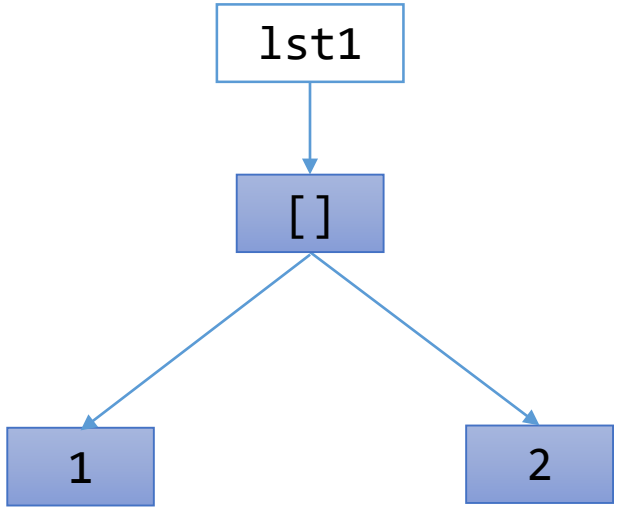
# Same Idea

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

- Output

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



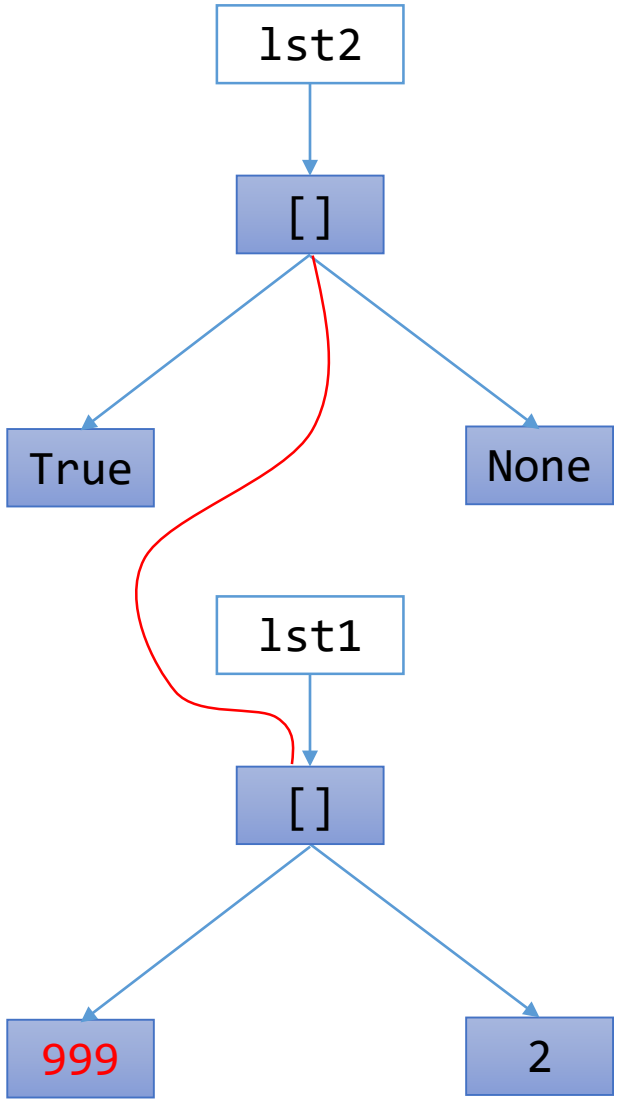



# Same Idea

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

- Output

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

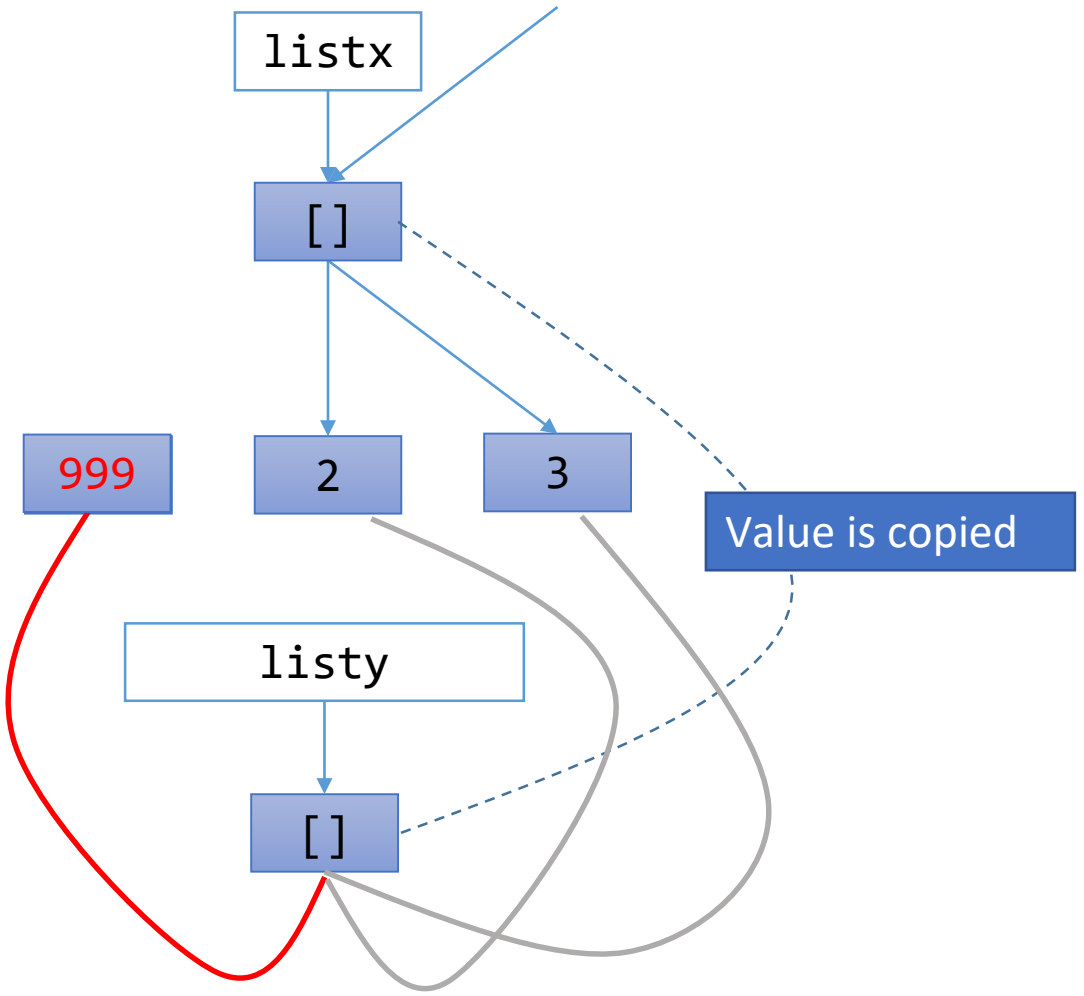


# How do we AVOID this?

- For list

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

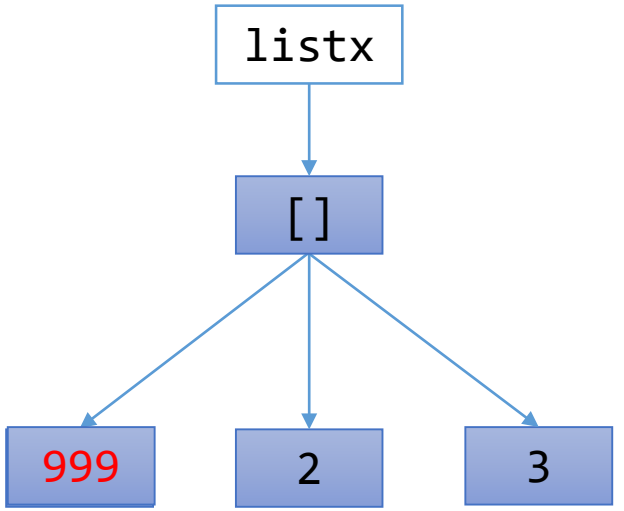
- We change `listx`
  - But `listy` is also changed?

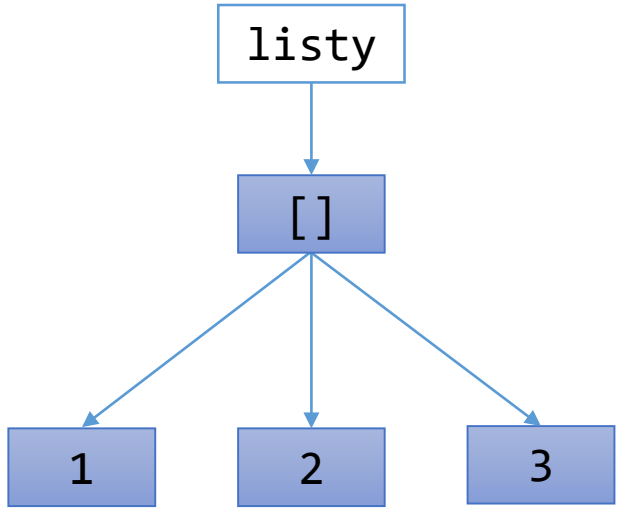


# Use function copy()

```
>>> 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



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

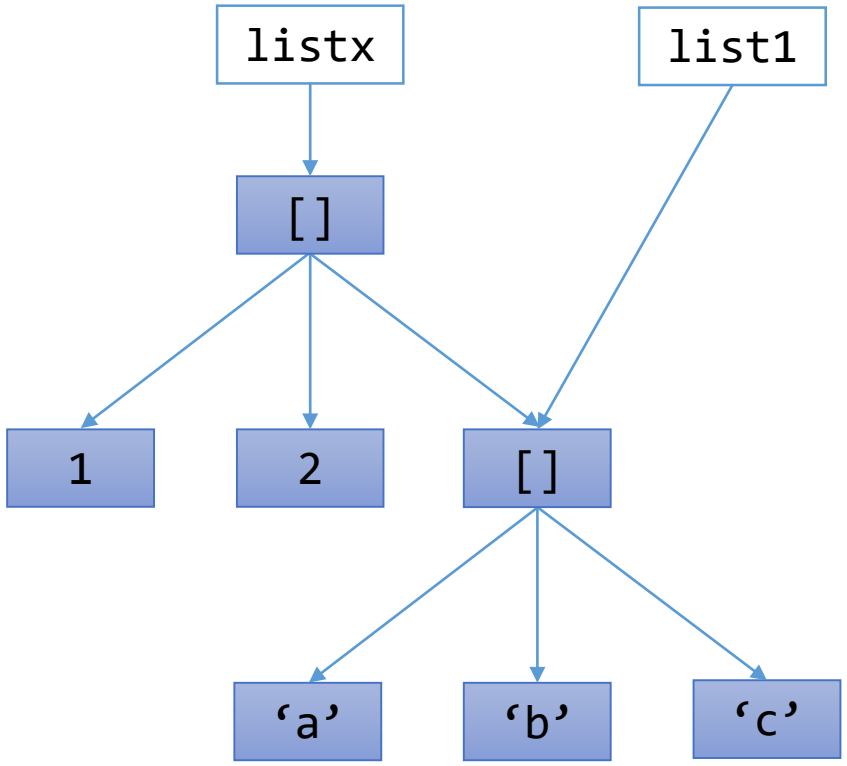
# However, life is not easy

```
>>> 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]
```

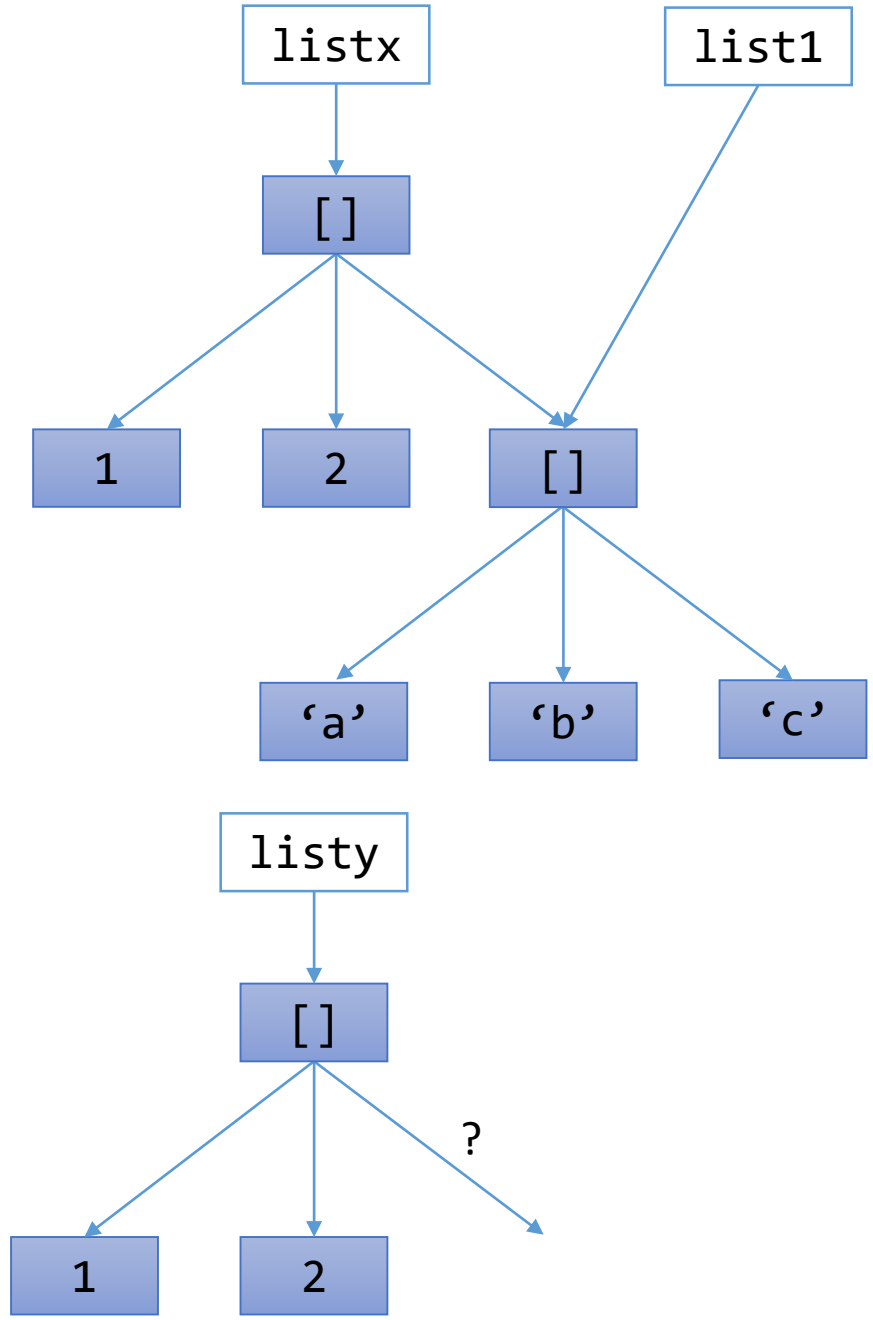
Did not use “copy()”



# 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']]
```

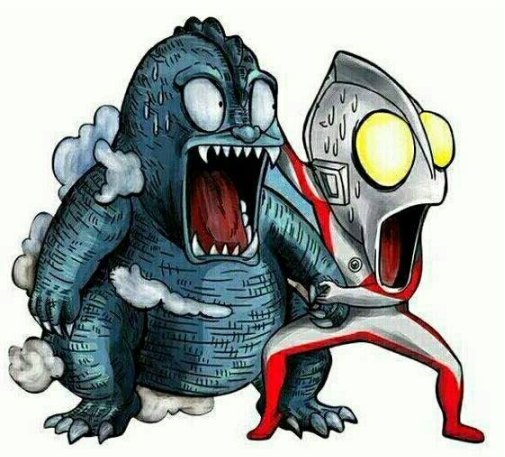
Even you use “copy()”

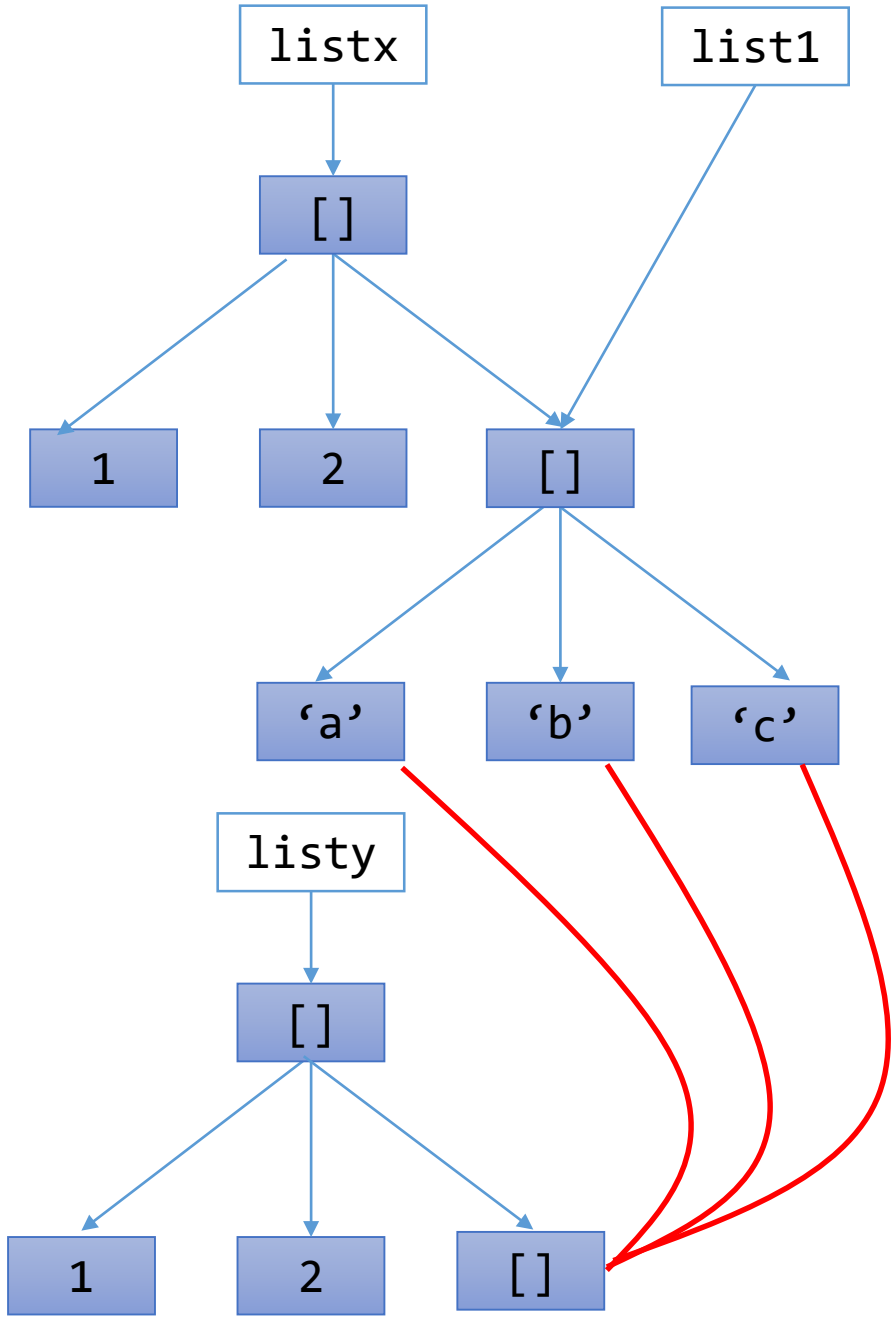


# Truth!

```
>>> 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 you use “copy()”



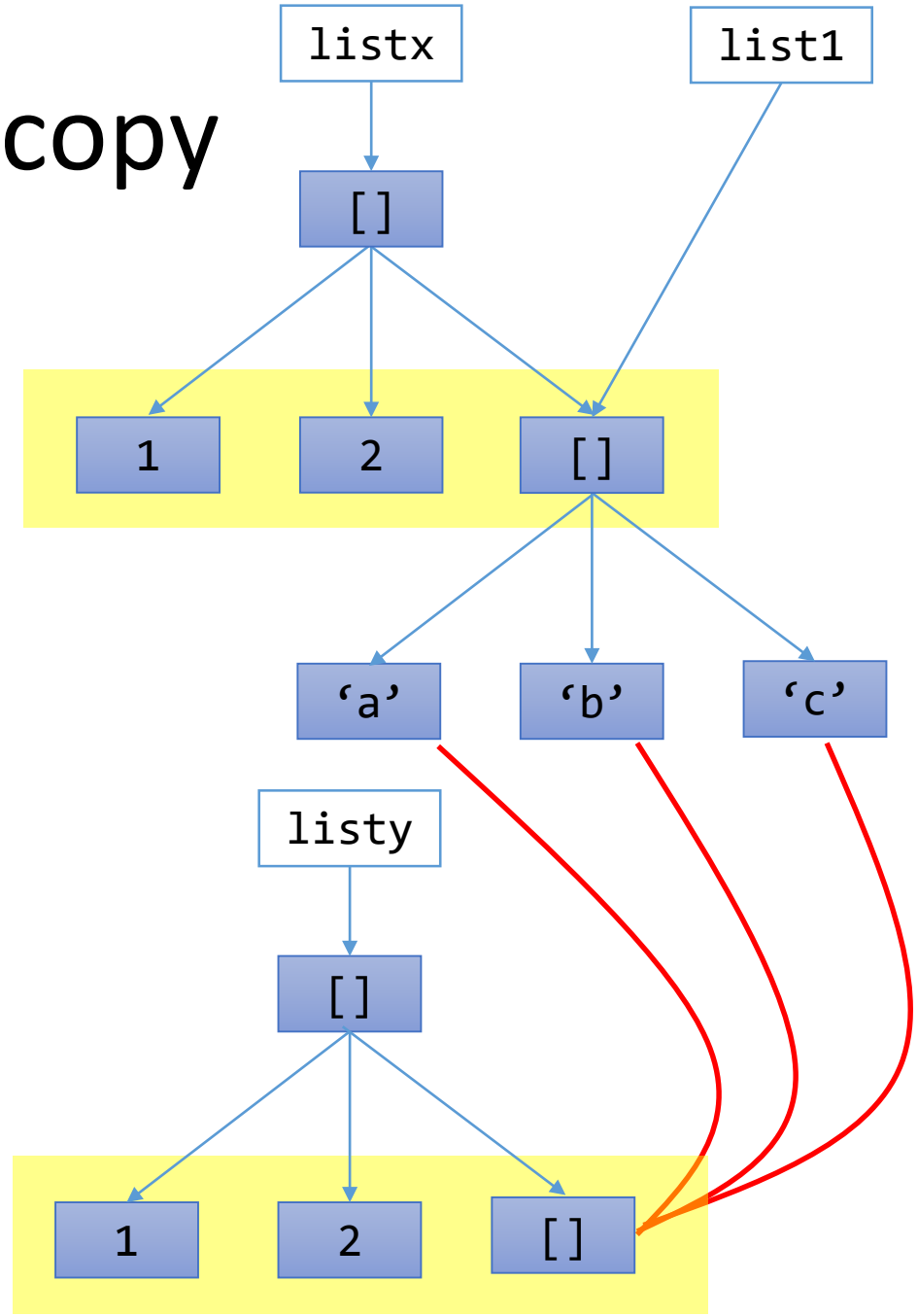


# “copy()” is only a SHALLOW copy

```
>>> 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 you use “copy()”

- Only **duplicate** the first layer



# Quick List Exercise

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

**Code**

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

**Output**

# Quick List Exercise

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

**Code**

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

**Output**

```
[1, [2], 3]
[1, [2], 3]
[1, [9], 3]
[1, [9], 3]
```

# Quick List Exercise

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

**Code**

```
print(lst_d == cpy_d)
print(lst_d is cpy_d)
print(lst_d[1] == cpy_d[1])
print(lst_d[1] is cpy_d[1])
```

**Output**

# Quick List Exercise

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

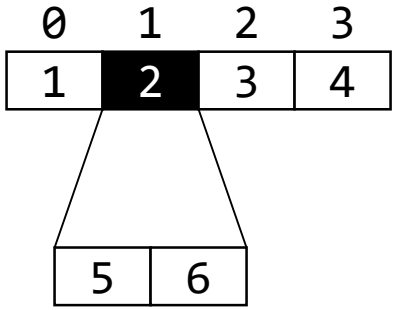
| **Code** | **Output** |
| --- | --- |
| `print(lst_d == cpy_d)` | `True` |
| `print(lst_d is cpy_d)` | `False` |
| `print(lst_d[1] == cpy_d[1])` | `True` |
| `print(lst_d[1] is cpy_d[1])` | `True` |

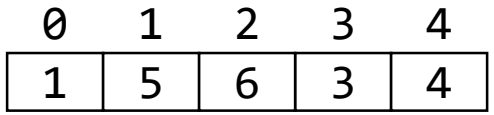
# Summary of List and “copy()”

- The list block diagram is essential to understand a very important concept in computer programming
  - Namely, pointers (memory address)
  - (Ask your seniors when they learned pointers in C)
- But for Python, this *hideous* concept is “well encapsulated” from beginners
- However, if you want to advance in programming, this is unavoidable
- Also, this give us more motivations to use tuples rather than lists
  - Because tuples do not have this complication

# List Mutation

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



# Exercises

Tuple and List

- We can use tuple/list to create a data that consists of mixed types.  In this exercise, you are hired by a fast food franchise to implement their automatic ordering system. One order can contain a lot of burgers. For example, my order for my family can be:

```
>>> myOrder
('BVPB', 'BCPCPB', 'BPCBPCB')
```

- So an “order” is a tuple of burger collections. You can download the file mealOrder.py to have a look at the implemented functions:

```
def makeEmptyOrder():
    return ()

def add_burgerToOrder(order,burger):
    return order + (burger,)
```

- And we can create an order like the followings

```
>>> myOrder = makeEmptyOrder()
>>> myOrder = add_burgerToOrder(myOrder,'BVPB')
>>> myOrder = add_burgerToOrder(myOrder,'BCPCPB')
>>> myOrder
('BVPB', 'BCPCPB')
```

# Tasks

- Write a function `enoughMoney(order, moneyInMyPocket)` to check if you have enough money to pay for your order, namely, returning `True` if `moneyInMyPocket` is more than or equal to the total price in the order, and `False` otherwise.

```
>>> enoughMoney(myOrder,15)
True
>>> enoughMoney(myOrder,13)
False
```

# Tasks

- Write a function `printReceipt(order)` to print a nice receipt for your order.

```
>>> printReceipt(myOrder)
Your orders:
BVPB $3.2
BCPCPB $5.6
BPCBPCB $6.1
--------------
Total: 14.9
```

# Tasks

- Write a function `removeItem(order,item)` to remove an item in the order.

```
>>> myOrder = removeOrder(myOrder,'BVPB')
>>> myOrder
('BCPCPB', 'BPCBPCB')
>>> myOrder = removeOrder(myOrder,'BVVVVVPPB')
The item BVVVVVPPB is not in the order.
```