Welcome to Data C88C!

Lecture 11: Mutability
Wednesday, July 9th, 2025
Week 3
Summer 2025
Instructor: Eric Kim (ekim555@berkeley.edu)

Announcements

- Maps project released!
- Reminder: weekly course surveys released on Gradescope
  - (small) amount of extra credit per survey filled out!

Important: Midterm logistics

- Midterm is next week!
  - **Required**: read this Ed post carefully: [link]
- "Main" exam time: **Tuesday July 15th, 3pm-5pm PST**
- Alternate Exams
  - Tuesday, July 15 7:00pm-9:00pm PT
  - Wednesday, July 16th, 8:20am-10:20am PT
- **Important**: if you can't make the "main" exam time, fill out the Google Form linked in the above Ed post!
- **(last resort)** if you can't make any of the midterm times, that's OK: your midterm score will be extrapolated based on your final exam score performance [link_syllabus]
- Midterm covers up to and including this Thursday, July 10th (Lecture 12: Object Oriented Programming)
-

Midterm logistics

- The midterm will be held over Zoom + Gradescope
- You must have your camera + screen sharing on during the entire exam, and we will be doing screen+camera recording.
- You must take the exam in a quiet room with no other students present
- Things to bring to the exam (and nothing else!):
  - **Photo ID**. Ideally your UCB student ID, but anything with your name + photo is fine, eg: Passport, driver's license, etc.
  - (Optional) Two (2) pages of handwritten (not typed!) notes
  - (Optional, recommended) Additional blank scratch paper, pencil/pen/eraser. Useful for drawing Env Diags!
  - We will also provide everyone with a 1-2 page digital PDF of additional reference
- Other than the above notes, the exam will be closed book, closed notes.
- For more details, read the Ed post: [link]

Midterm study tips

- Do LOTS of previous exams: https://c88c.org/su25/resources/
- Practice the Gradescope timed online exam format: "(Optional) Practice Online Midterm (SU24)"
- Participate in all course content
  - Watch all lecture videos (including Prof. John DeNero's YouTube videos)
  - Attend (or watch recorded) lab sections
  - Complete (and understand!) the labs and homework assignments
  - Read the course textbook to reinforce concepts / fill any gaps
- Practice, practice, practice
  - Tip: there is value in re-doing coding exercises / previous tricky HW/lab assignments!

Lecture Overview

- Mutability
  - List mutation
  - Pure vs impure functions
- Identity
  - `is` vs `==`

List mutation

- In Python, we can modify (mutate) lists directly via special methods like `lst.append()`

```
>>> my_lst = [1, 2, 3]
>>> my_lst.append(42)
>>> my_lst
[1, 2, 3, 42]
>>> my_lst[0] = 9
my_lst
[9, 2, 3, 42]
>>> my_lst.pop()
42
>>> my_lst
[9, 2, 3]
```

List mutation (reference)

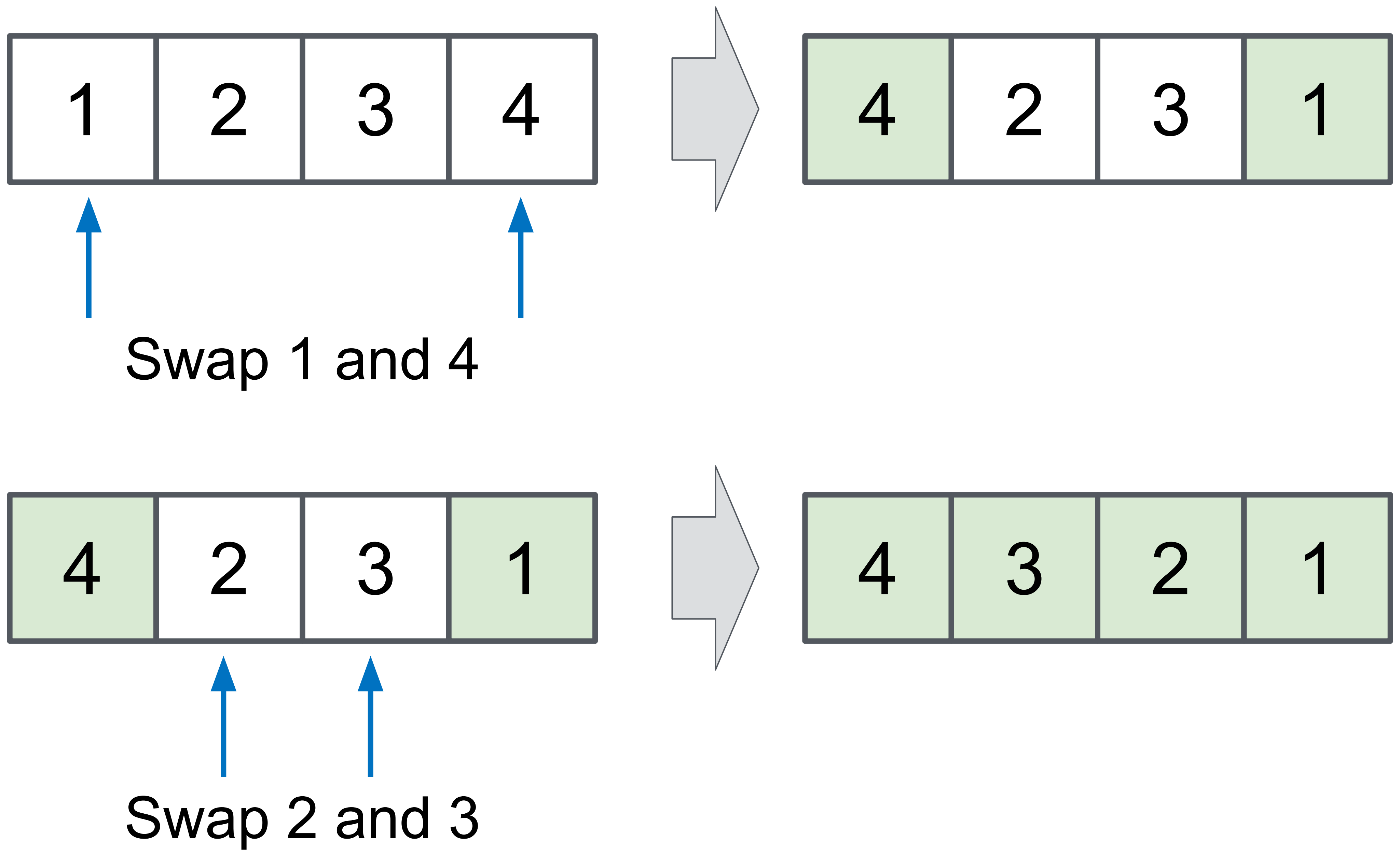
Operation	Result
`lst[i] = new_value`	Assign `new_value` to list at index `i`. If `i` is out of bounds, then raise "IndexError: list assignment index out of range".
`lst.append(new_value)`	Add new value to the end of the list
`lst.extend(other_lst)`	Add multiple values (other_seq) to end of seq. Similar: `lst += other_lst`
`lst.pop(ind)`	Remove (and return) last element of list. If `ind` is omitted, this implicitly sets `ind=-1`.
`lst.remove(elem)`	Remove the first instance of `elem` in the list. If `elem` is not in list, raises: "ValueError: x not in list"

Example: reverse_lst()

Question: write a function `reverse_lst(lst)` that, given an input list, reverses the list via direct modification ("in-place").

```
>>> nums = [1, 2, 3, 4]
>>> reverse_lst(nums)
>>> nums
[4, 3, 2, 1]
```

Hint: swap the first and last values, then repeat going "inwards"




```
def reverse_lst(lst):
    """Given an input list, reverse the list
    via direct modification ("in-place").
    >>> nums = [1, 2, 3, 4]
    >>> reverse_lst(nums)
    >>> nums
    [4, 3, 2, 1]
    """
    for i1 in range(len(lst) // 2):
        i2 = len(lst) - 1 - i1
        tmp = lst[i1]
        lst[i1] = lst[i2]
        lst[i2] = tmp
```

Question: using negative indexes, what should `i2` be?

Answer: `i2 = -(i1 + 1)`

Caution: mutation vs non-mutation

- Be sure to understand the difference between functions that modify (mutate) their inputs, and functions that create something new (eg returns a new list, etc).

```
def reverse_lst(lst):
    for i1 in range(len(lst) // 2):
        i2 = len(lst) - 1 - i1
        tmp = lst[i1]
        lst[i1] = lst[i2]
        lst[i2] = tmp
```

```
def reverse_lst_v2(lst):
    # slicing creates a new list
    return lst[::-1]
```

```
>>> nums = [1, 2, 3, 4]
>>> reverse_lst(nums)
>>> nums
[4, 3, 2, 1]
```

```
>>> nums = [1, 2, 3, 4]
>>> nums_r = reverse_lst_v2(nums)
>>> nums_r
[4, 3, 2, 1]
>>> nums
[1, 2, 3, 4]
```

original `nums` is still the same, even after calling `reverse_lst_v2()`!

Functions and mutation

- Common convention: functions that modify its inputs often return `None` (ie have no return statement)
  - Their side effect is the functions "output"
  - Aka **"impure"** functions
- On the other hand, functions that don't modify their inputs return a new value
  - Aka **"pure"** functions

Aka the function modifies the inputs **"in place"**

```
def square_nums_mutate(nums):
    i = 0
    while i < len(nums):
        nums[i] = nums[i] ** 2
        i += 1

>>> nums1 = [1, 2, 3]
>>> square_nums_mutate(nums1)
>>> nums1
[1, 4, 9]
```

Here, `nums1` is modified!

```
def square_nums_pure(nums):
    return [n ** 2 for n in nums]

>>> nums1 = [1, 2, 3]
>>> square_nums_pure(nums1)
[1, 4, 9]
>>> nums1
[1, 2, 3]
```

Here, `nums1` is NOT modified. Instead, `square_nums_pure()` created a new list

```
def square_nums_alt(nums):
    out = []
    for n in nums:
        out.append(n ** 2)
    return out
```

Question: is `square_nums_alt()` a pure function, or a non-pure function?

Answer: it's a pure function! Even though there is mutation going on in the function body (`out.append()`), to the "outside world" `square_nums_alt()` is a pure function (not modifying input args / global state)

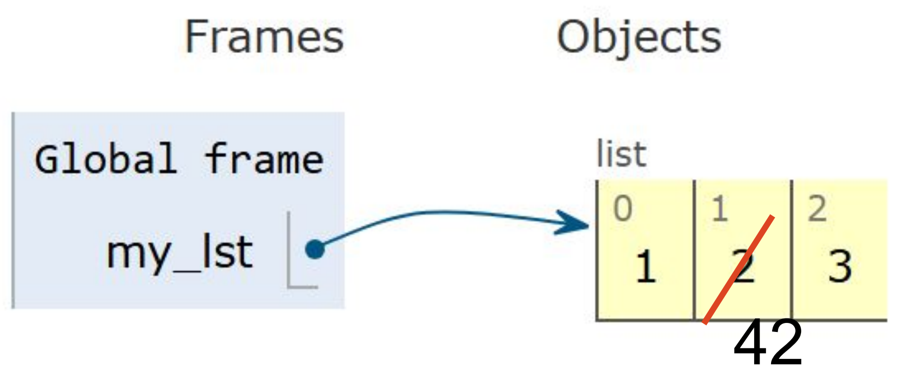
Why care about pure vs impure functions?

- Generally speaking, pure functions are easier to understand and reason about
- Ex: calling the function with the same arguments always has the same exact behavior, no matter how often you call the function and in what order you call it
- Some programming languages embrace pure functions: functional programming
  - Examples: Lisp (eg Scheme), Haskell, ML (the programming language, not "machine learning")
- In practice: most main-stream programming languages allow impure functions, and leave it to the programmer to adopt functional-style programming if they wish to do so
  - Examples: Python, Java, C/C++
- Aka "programming paradigms"

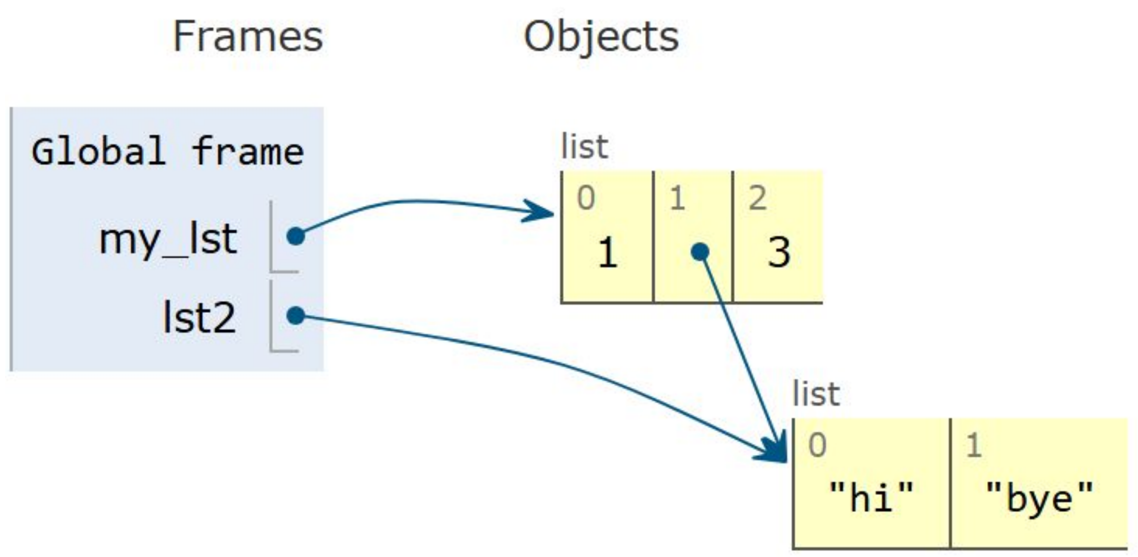
Environment Diagrams: list mutation

- `lst[i] = new_val`
  - If `new_val` is a **primitive value** (eg int): replace index `i` contents
  - If `new_val` is a **compound value** (eg another list): draw an arrow to `new_val`
- `append()`/`extend()`: Add additional boxes (entries) to the list

```
my_lst = [1, 2, 3]
my_lst[1] = 42
```

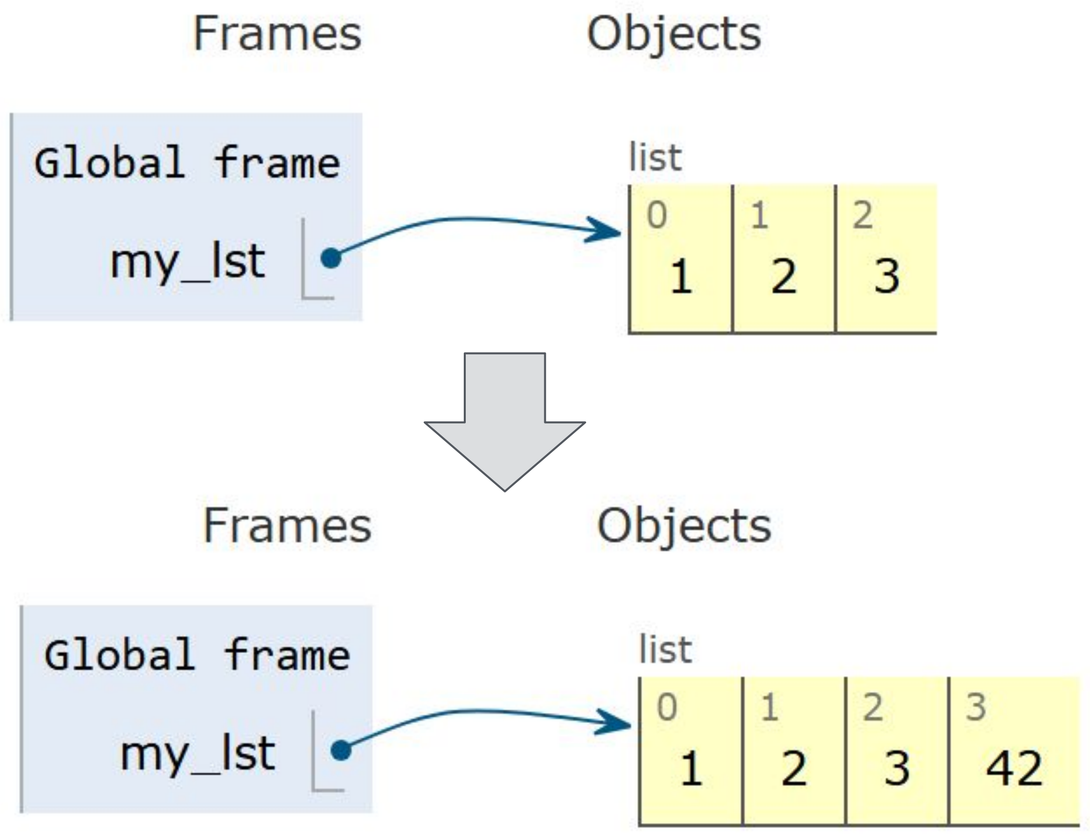



```
my_lst = [1, 2, 3]
lst2 = ['hi', 'bye']
my_lst[1] = lst2
```



Python 3.6
(known limitations)

```
1  my_lst = [1, 2, 3]
2  my_lst.append(42)
```



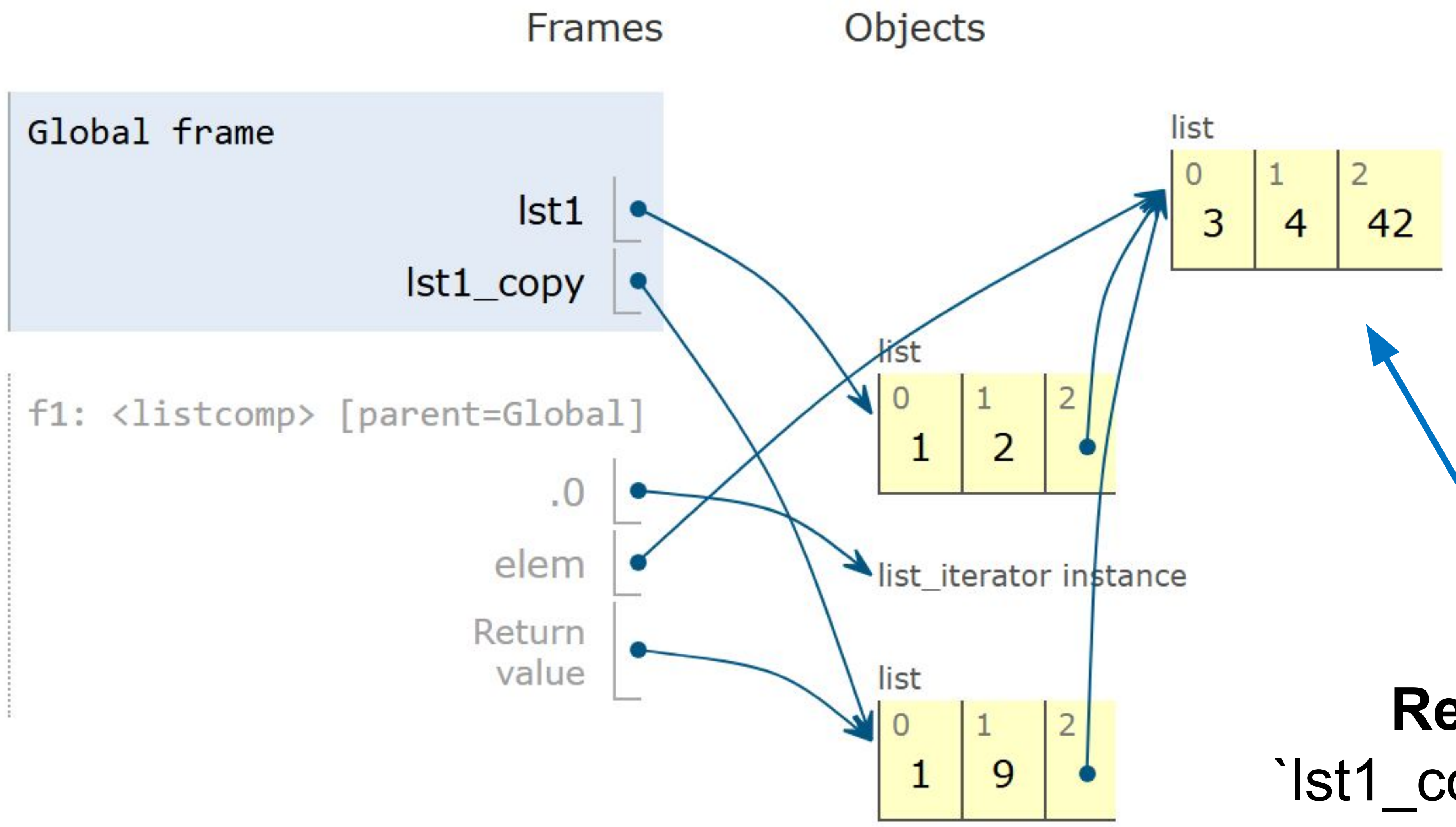
Shallow vs deep copy

- Recall that lists can contain other lists. A list is an example of a "compound" object.

This is known as a "shallow" copy.

PythonTutor: [link]

```
>>> lst1 = [1, 2, [3, 4]]
# Make a copy of lst1
>>> lst1_copy = list(lst1)
>>> lst1_copy[1] = 9
>>> lst1_copy
# BLANK_A
>>> lst1
# BLANK_B
>>> lst1_copy[2].append(42)
>>> lst1_copy
# BLANK_C
>>> lst1
# BLANK_D
```



Reason: `lst1[2]` and `lst1_copy[2]` both point to the same underlying list object. Thus, mutations to `lst1[2]` propagate to `lst1_copy[2]` (and vice-versa)

A change to `lst1_copy` modified `lst1`!

Question: what does Python display?

Answer:

```
BLANK_A: [1, 9, [3, 4]]
BLANK_B: [1, 2, [3, 4]]
BLANK_C: [1, 2, [3, 4, 42]]
BLANK_D: [1, 2, [3, 4, 42]]
```

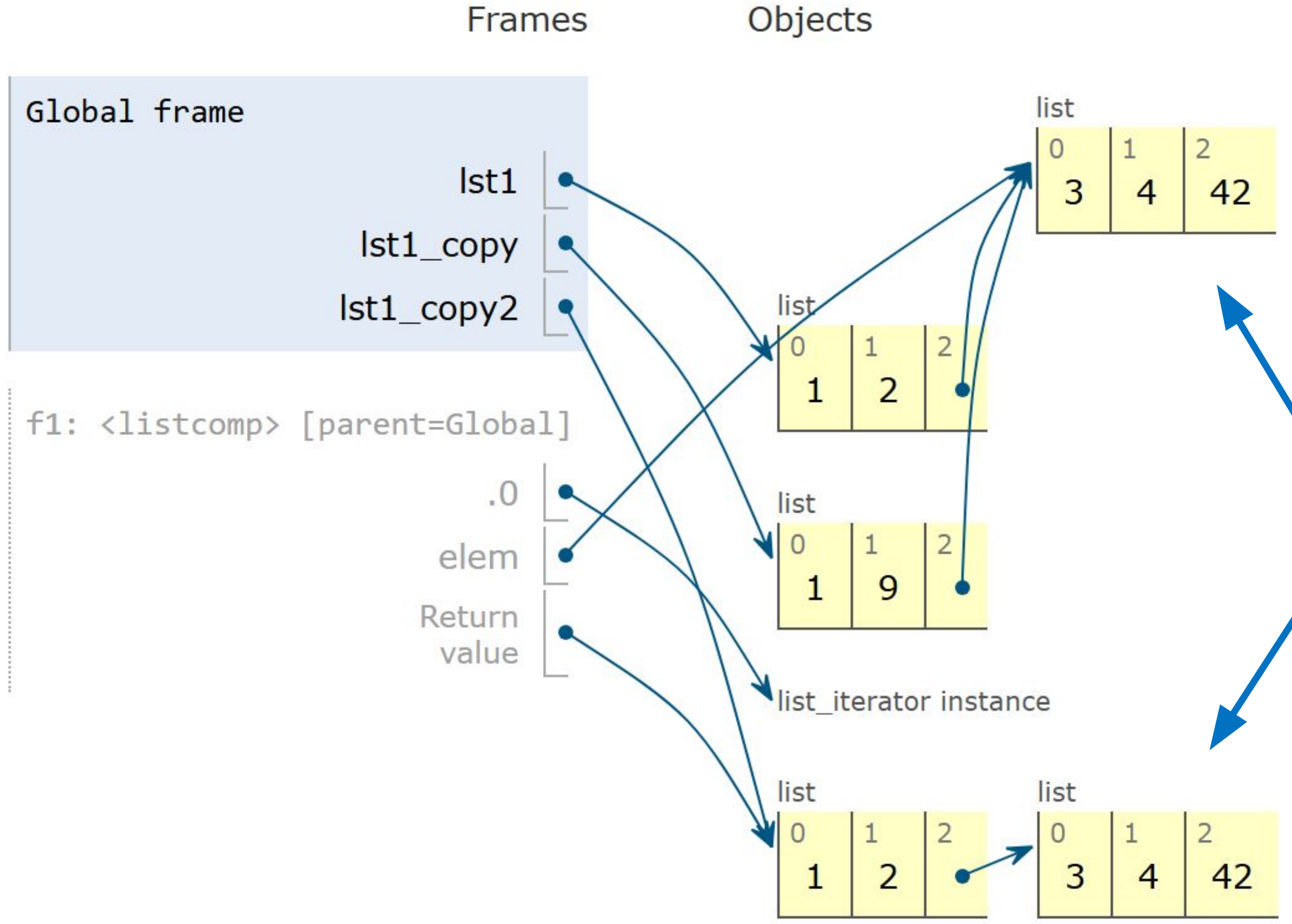
Shallow vs deep copy

- **Definition**: "A **shallow copy** constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original." [link_source]
  - ie: A shallow copy only copies the "first level" of the list

```
>>> lst1 = [1, 2, [3, 4]]
# Make a copy of lst1
>>> lst1_copy = [elem for elem in lst1]
```

This is known as a "shallow" copy.

PythonTutor: [link]



Success: `lst1_copy2` is detached from `lst1`!

Question: how could we make a "deeper" copy of `lst1`?
Hint: the `list(lst)` constructor creates a (shallow) copy of `lst`.
Hint: to tell if something is a list, use `type(x) == list`*

```
# using list comprehension and conditional expression
>>> lst1_copy2 = [list(elem) if (type(elem) == list) else elem for elem in lst1]
```

Issue: this doesn't work if there is a list at the "third" level, ex:

```
>>> lst2 = [1, 2, [3, [4]]]
```

Let's generalize this "copy deeper" idea into: **"deep copy"**

* **Aside**: one can also do `isinstance(x, list)`. If you're curious, here is a long discussion on why one should use `type()` vs `isinstance()`: [link]

Deep copy

- **Definition**: "A **deep copy** constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original." [link_source]
  - ie if the deep copy encounters a compound object (that may contain other compound objects), deep copy will **recursively copy** the compound object.

Question: implement the `deep_copy(lst)` function that, given an input list, deep-copies the list.
Hint: implement it recursively!

```
>>> lst = [1, 2, [3], [4, [5, 6]]]
>>> lst_deepcopy = deep_copy(lst)
# prove that the copy is deep, not shallow
>>> lst_deepcopy[3][1].append(42)
>>> lst_deepcopy
[1, 2, [3], [4, [5, 6, 42]]]
>>> lst
[1, 2, [3], [4, [5, 6]]]
```

```
def deep_copy(lst):
    if not lst:
        return []
    elif type(lst[0]) == list:
        return [deep_copy(lst[0])] + deep_copy(lst[1:])
    else:
        # lst[0] is a primitive object (eg int)
        return [lst[0]] + deep_copy(lst[1:])
```

Approach: if the first element of `lst` is a list, then `deep_copy(lst[0])`, and concatenate it to the result of `deep_copy(lst[1:])`.
Easy case: if the first element is a primitive (eg int), then concatenate [lst[0]] to `deep_copy(lst[1:])`.

Shallow copy vs Deep copy

Observe how similar these two implementations are!

```
def deep_copy(lst):
    if not lst:
        return []
    elif type(lst[0]) == list:
        return [deep_copy(lst[0])] + deep_copy(lst[1:])
    else:
        # lst[0] is a primitive object (eg int)
        return [lst[0]] + deep_copy(lst[1:])
```

```
def shallow_copy(lst):
    if not lst:
        return []
    else:
        return [lst[0]] + shallow_copy(lst[1:])
```

```
# Other ways to create shallow copies of lists
lst_scopy1 = list(lst)
lst_scopy2 = lst[:]  # slicing creates shallow copy
lst_scopy3 = [x for x in lst]

lst_scopy4 = []
for x in lst:
    lst_scopy4.append(x)

lst_scopy5 = []
lst_scopy5.extend(lst)
```

Mutation and Identity

Sameness and Change

- As long as we never modify objects, a compound object is just the totality of its pieces
- This view is no longer valid in the presence of change
- A compound data object has an "identity" in addition to the pieces of which it is composed
- A list is still "the same" list even if we change its contents
- Conversely, we could have two lists that happen to have the same contents, but are different

```
>>> a = [10]
>>> b = a
>>> a == b
True
>>> a.append(20)
>>> a
[10, 20]
>>> b
[10, 20]
>>> a == b
True
```

```
>>> a = [10]
>>> b = [10]
>>> a == b
True
>>> b.append(20)
>>> a
[10]
>>> b
[10, 20]
>>> a == b
False
```

Identity Operators

Identity

<exp0> **is** <exp1>

evaluates to True if both <exp0> and <exp1> evaluate to the same object

Equality

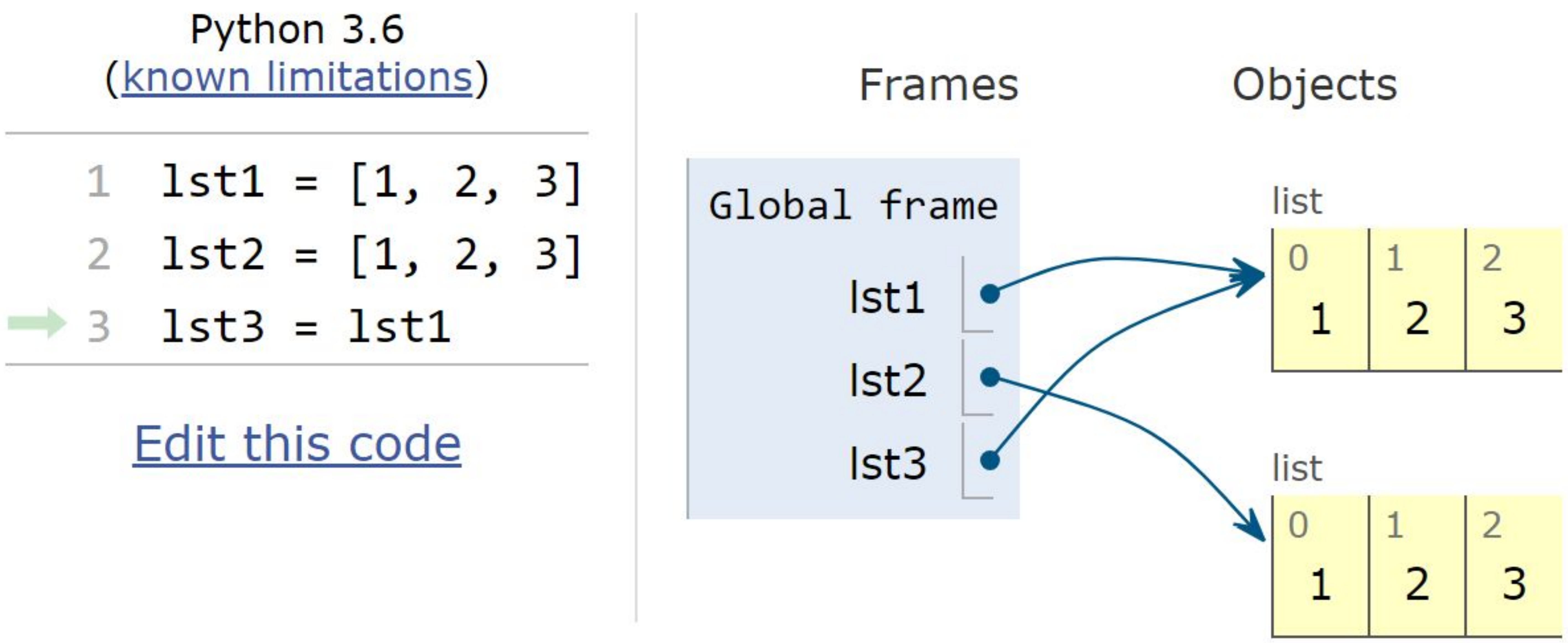
<exp0> **==** <exp1>

evaluates to True if both <exp0> and <exp1> evaluate to equal values

Identical objects are always equal values

(Demo)

Equality (`==`) vs identity (`is`)



On the other hand:

```
>>> lst1 == lst2
True
>>> lst1 is lst2
False
```

Although `lst1` has "**value equality**" to `lst2`, they point to different objects (have **different identities**)

lst1 and lst2 point to different objects, but have the same values

=> Equality (lst1 == lst2)

Aka “Value equality”

lst1 and lst3 point to the same object

=> Identity (`lst1 is lst3`)

Aka “Identity equality”

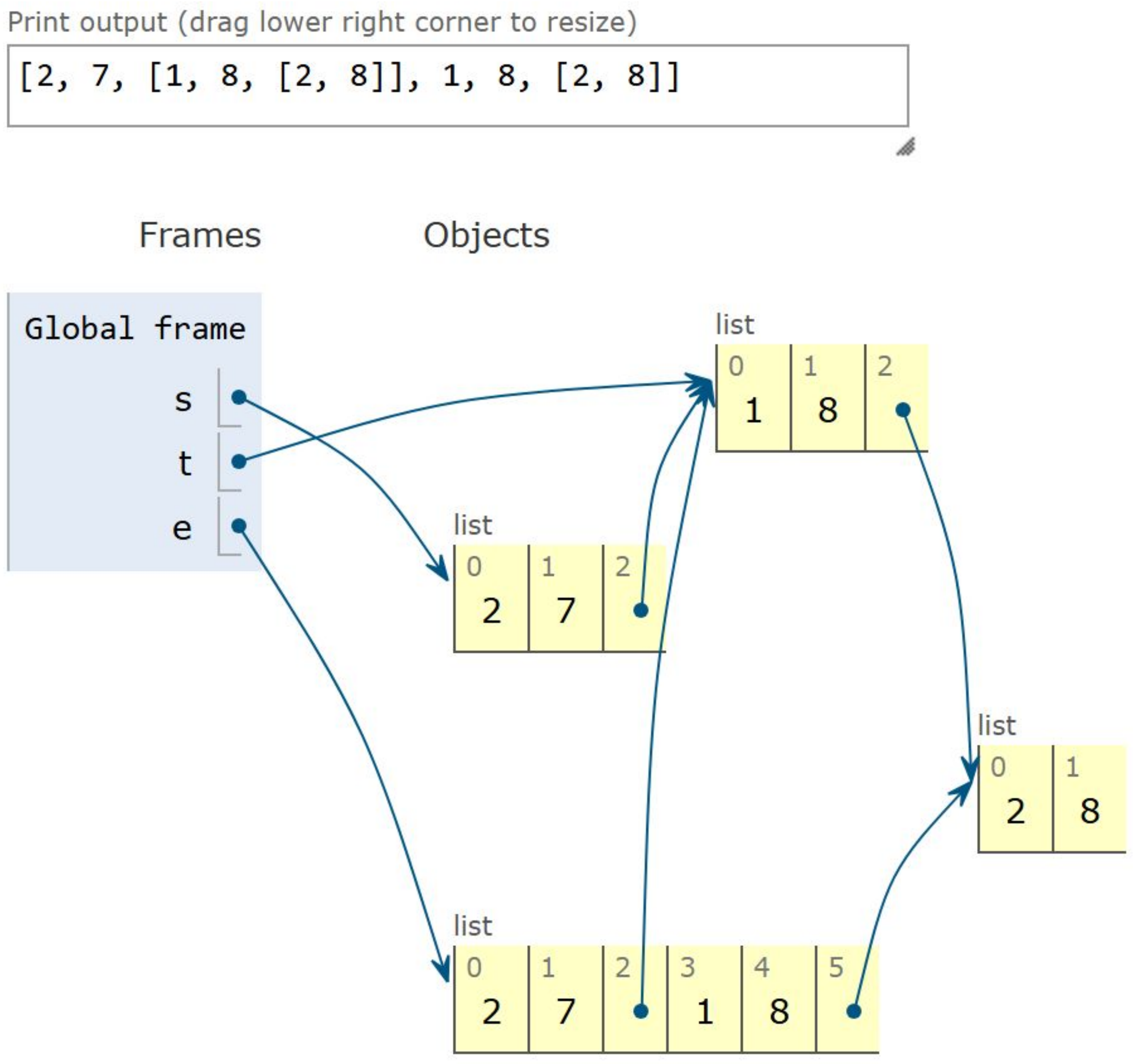
Mutation and Names

If multiple names refer to the same mutable object (directly or indirectly), then a change to that object is reflected in the value of all of these names.

Question: What numbers are printed (and how many of them)?

```
s = [2, 7, [1, 8]]
t = s[2]
t.append([2])
e = s + t
t[2].append(8)
print(e)
```

Answer: `[2, 7, [1, 8, [2, 8]], 1, 8, [2, 8]]`



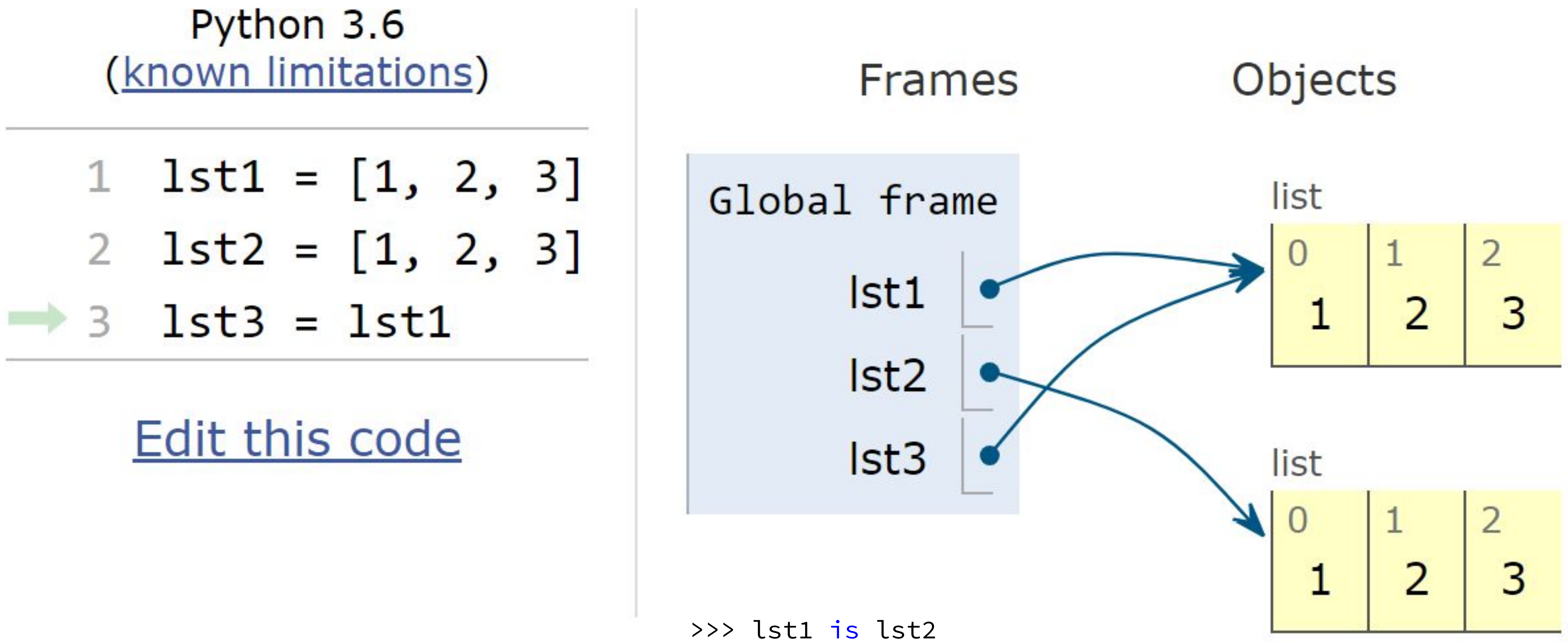
PythonTutor: [link]

https://pythontutor.com/cp/composingprograms.html#code=s%20%3D%20%5B2,%207,%20%5B1,%208%5D%5D%0At%20%3D%20s%5B2%5D%0At.append%28%5B2%5D%29%0Ae%20%3D%20s%20%2B%20t%0At%5B2%5D.append%288%29%0Aprint%28e%29%0A&cumulative=true&curInstr=6&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

(Aside) How does Python implement the `is` operator?

How does Python implement the `is` operator?

Intuitive (visual) answer: `a is b` is True iff `a` points to the same exact object as `b` points to.




```
>>> lst1 is lst2
False
>>> lst1 is lst3
True
```

(Aside) How does Python implement the `is` operator?

How does Python implement the `is` operator?

Technical answer: `a is b` is True iff `a` points to the same **memory address** as `b` points to.

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
>>> lst3 = lst1
>>> id(lst1)
1906951142144
>>> id(lst2)
1906951128448
>>> id(lst3)
1906951142144
```

These are memory addresses! Eg somewhere in your CPU RAM

Tip: think of RAM as a long array/list, and an address as an index into the list.

```
>>> lst1 is lst2
False
>>> lst1 is lst3
True
>>> id(lst1) == id(lst2)
False
>>> id(lst1) == id(lst3)
True
```

Thus, the `is` operator is actually comparing memory addresses behind the scenes.

(Aside) Python's `id()` function

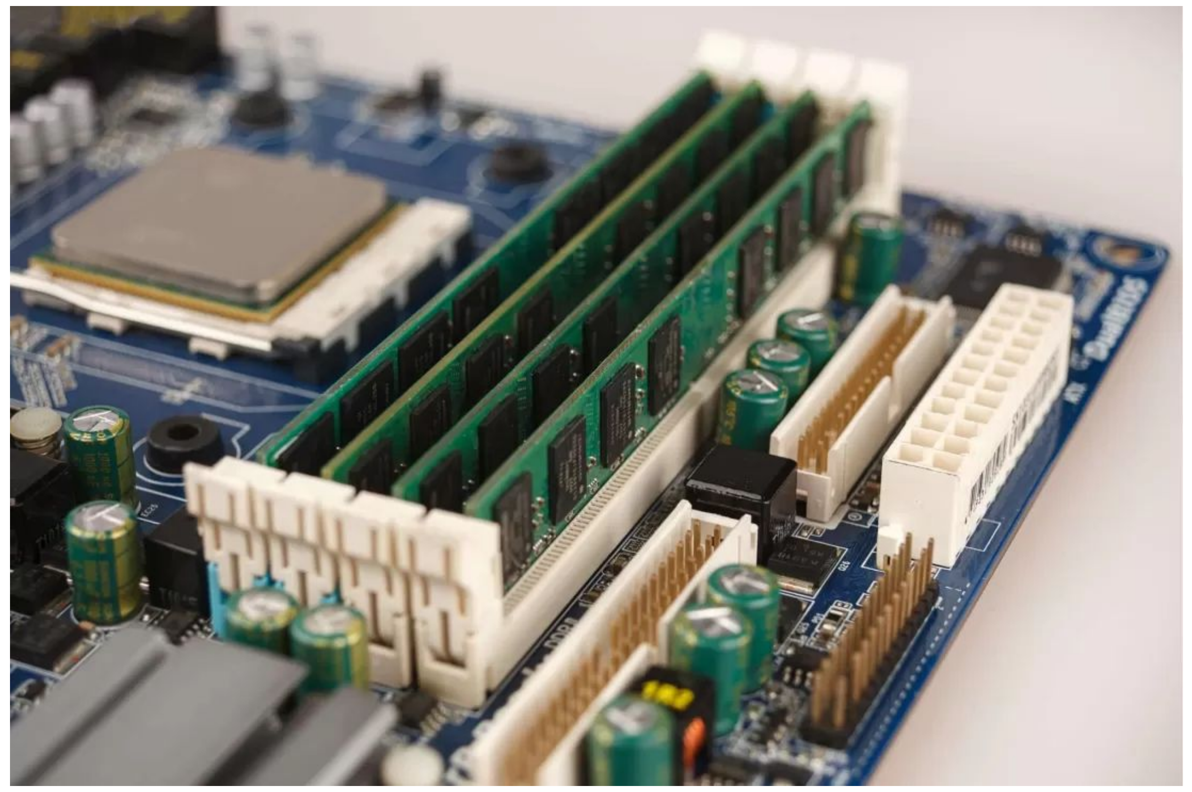
In Python, each object has a function `` `id()` `` which returns its "**memory address**"
CPU memory (random access memory, aka "RAM")
Example: In 2024, a Macbook Pro 14" has 8 GB - 36 GB CPU RAM
Every Python object (eg list, string, etc) lives somewhere in your CPU memory.

- Aside: GPU (graphical processing units) have their own separate GPU memory. State-of-the-art ML models (like ChatGPT, etc) are notoriously GPU-memory intensive

Bill Gates once said* in 1981 "640K of memory should be enough for anybody."

- * Not actually true, but it makes a funny story

Related courses: CS61C (Architecture), CS162 (Operating Systems), CS164 (Programming Languages and Compilers)



Credit: https://www.techspot.com/article/2024-anatomy-ram/