CSc 120
Introduction to Computer Programming II

02: References

Data organization

The stuff under the hood

How are data values organized?

Python 3.5.2 Shell

File Edit Shell Debug Options Window Help

```
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:1
8) [MSC v.1900 32 bit (Intel)]
Type "copyright", "credits" or "license()" for more inf
ormation.
>>> x = [1,2,3,4,5]
>>> x
[1, 2, 3, 4, 5]
>>> x[2]
3
>>> x[2] = [10,20,30,40,50,60,70,80,90,100]
>>>
>>> x
[1, 2, [10, 20, 30, 40, 50, 60, 70, 80, 90, 100], 4, 5]
>>> x[2][5]
60
>>>
```

what exactly does this do?

how exactly is x's value found?
how about that of x[2]?

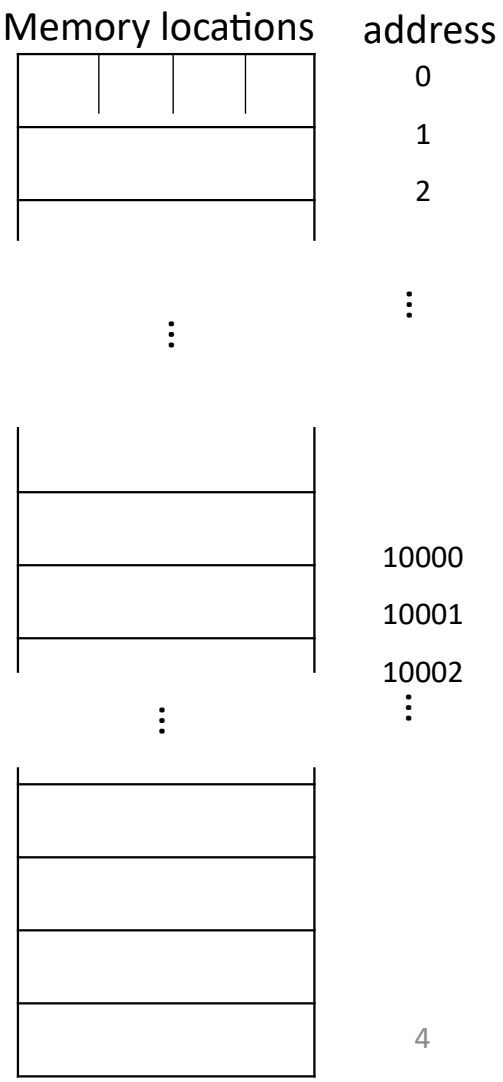
That's a lot of numbers! How can they all fit into x[2]?

how does this work?

Data organization in memory

Computer memory is organized as a sequence of *locations*

- each location is identified by its *address* (a number)
- a location typically consists of 8 bits (a "byte")
- bytes are often grouped into "words" (32 or 64 bits)



Data organization in memory

A location has only a fixed (limited) number of bits (32 or 64)

$\Rightarrow$ it can only hold a fixed (limited) amount of data

```
>>> x = [10,20,30,40,50,60,70,80,90,100]
```

the variable x is at a location
$\Rightarrow$ fixed (limited) capacity

but this value can be arbitrarily big: e.g., a million, or a billion, or a trillion elements

How can we make this work?

Object identity and References

Intuition behind the solution

- Use a "name" for each value computed
  - real-world analogy: names that designate entities, e.g., "Franklin D. Roosevelt", "the car with AZ plate# MTP-242"
  - "names" in the computer: a sequence of 0s and 1s
    - looks like a number
- When executing a statement like

  x = *value*
  - create/compute *value*
  - store its name in x
- When accessing the value of x: use the name stored to look up the value

Object identity

- Every data value in the Python system has a unique *identity number* (≈ its "name")
  - the identity no. of a value v is given by id(v)
  - id#s correspond to the intuition of the "name" of a value
- When an assignment `x = [10,20,30]` is executed:
  - the list [10, 20, 30] is put somewhere in memory
  - the list's identity number is stored in x
    - this is called a *reference* to the list
- To find the value of x, we use this reference to locate and retrieve its value

What happens during execution

Code executed	Actions within Python system
>>> x = [10, 20, 30]	1. Construct the list [10, 20, 30] somewhere in memory 2. Store *a reference to this list* in x. I.e.: • retrieve the id# of this list • store this id# in x.
>>> x [10, 20, 30]	1. Find the id# stored in x 2. Retrieve the value associated with that id# 3. Print out this value

Data structure diagrams

Diagramming data structures

- Usually, the exact numerical value of an id() or a reference is not important
  - what matters is the *refers-to* relationship, i.e., what refers to what
- We can show such relationships graphically:
  - "x is a reference to *value*" (equivalently: "x refers to *value*") is shown as

$$x \longrightarrow value$$

Data structure diagrams: Values I

Value	Actions in the Python system	Diagram
a number: 123	Construct the value 123	123
a string: "abc"	1. Construct the string "abc" 2. Use its id# to refer to it	"abc"

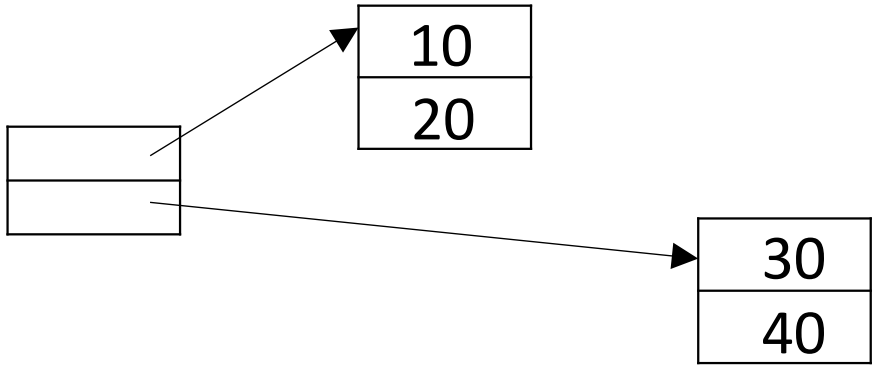
Data structure diagrams: Values II

Value	Actions in the Python system	Diagram
a list: [10, 20, 30]	1. Construct a "container" of appropriate size (i.e., a set of locations for holding values) 2. Store the individual list elements (10, 20, etc.) in successive slots in this container 3. Use the container's id# to refer to the list	container 10 20 30

Nested values

- A Python list is just a container with a single sequence of references
- For nested lists:
  - construct the values of the individual elements of the list
  - construct a container for the outer list
  - store references to the list elements in the outer container

E.g.: [[10, 20], [30, 40]] :

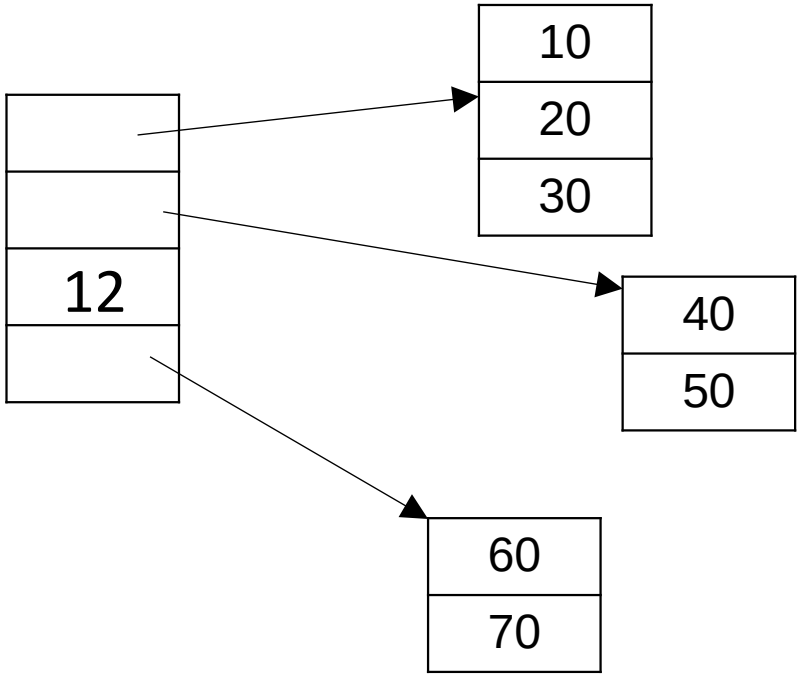


An example

Value:

[[10, 20, 30], [40, 50], 12, [60, 70]]

Diagram:



EXERCISE

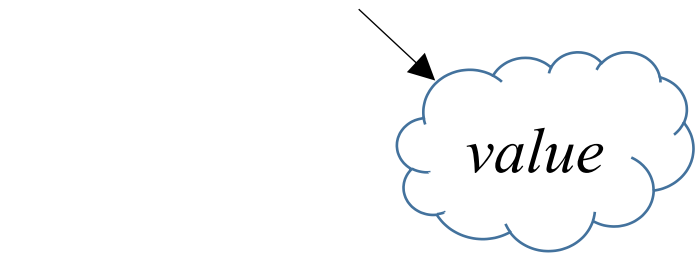
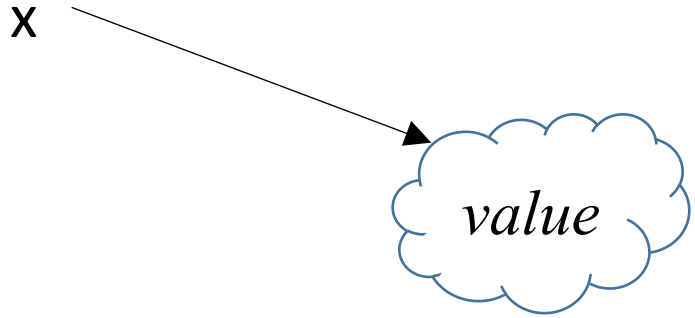
Draw the data structure diagram for the following:

[[1, 0, 0], [0, 1, 0], [0, 0, 1]]

Data structure diagrams: assignment

Handling an assignment x = *value*

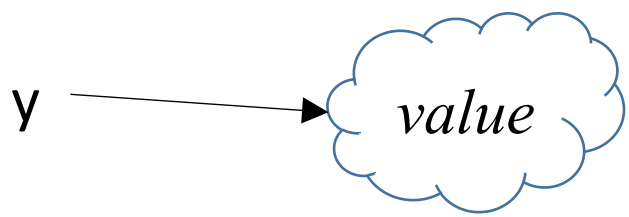
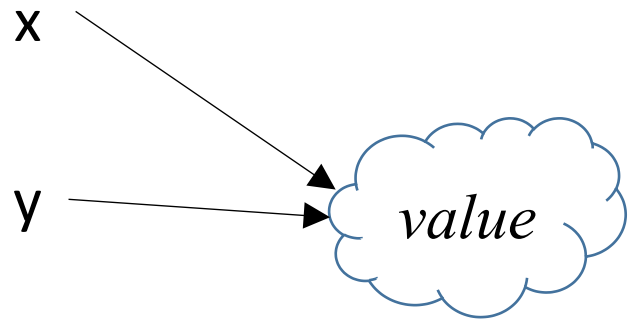
- get the reference to *value* (compute/create *value* if necessary)
- store this reference in x

Before	After
	

Data structure diagrams: assignment

Example

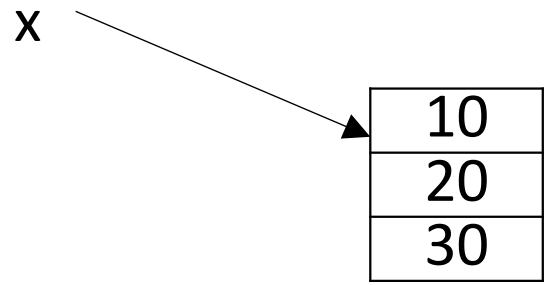
x = y

Before	After
	

Data structure diagrams: assignment

Example

x = [10, 20, 30]

Before	After
x	

EXERCISE

```
>>> x, y = "ab", 12
```

← *what are the diagrams for x and y?*

```
>>> z = [x, y]
```

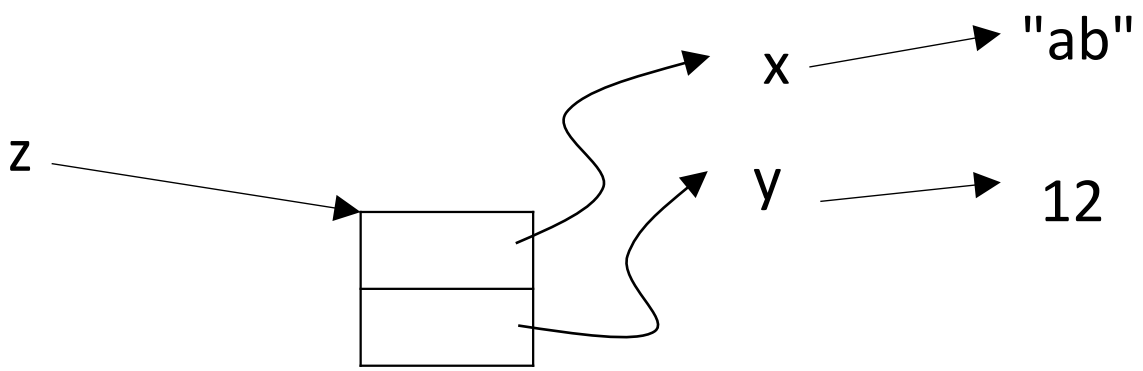
← *what is the diagram for z?*

```
>>> w = z[1]
```

← *what is the diagram for w?*

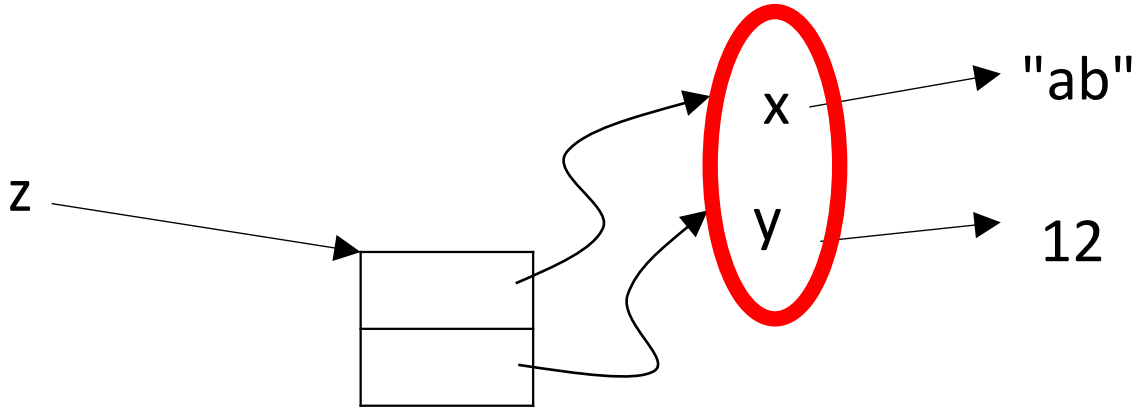
```
>>> x, y = "ab", 12
>>> z = [x, y]
```

What is wrong with this diagram???

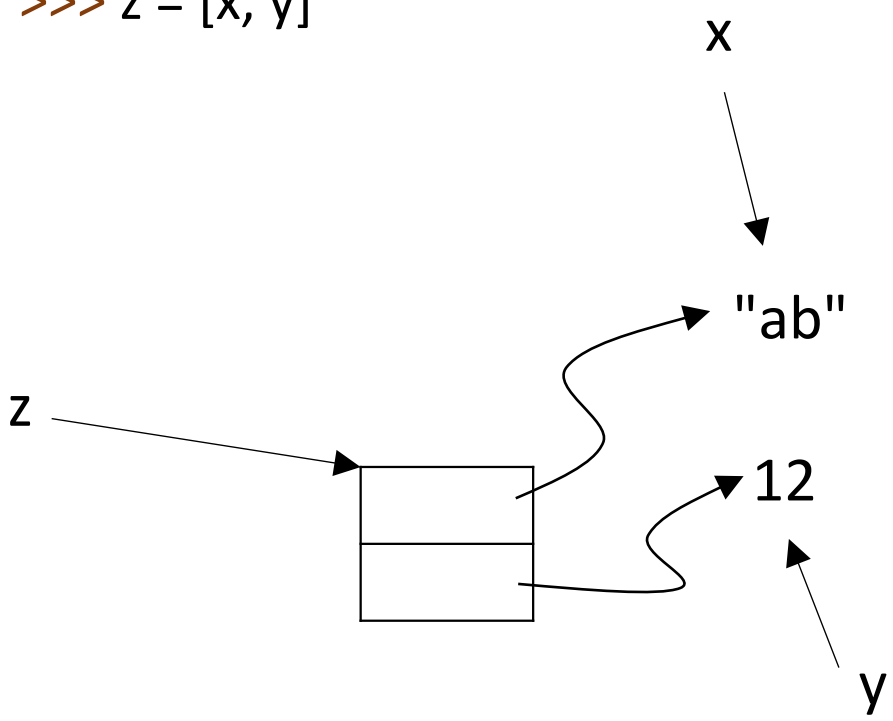


```
>>> x, y = "ab", 12
>>> z = [x, y]
```

References never point at **variables.** References point at **objects.**



```
>>> x, y = "ab", 12
>>> z = [x, y]
```

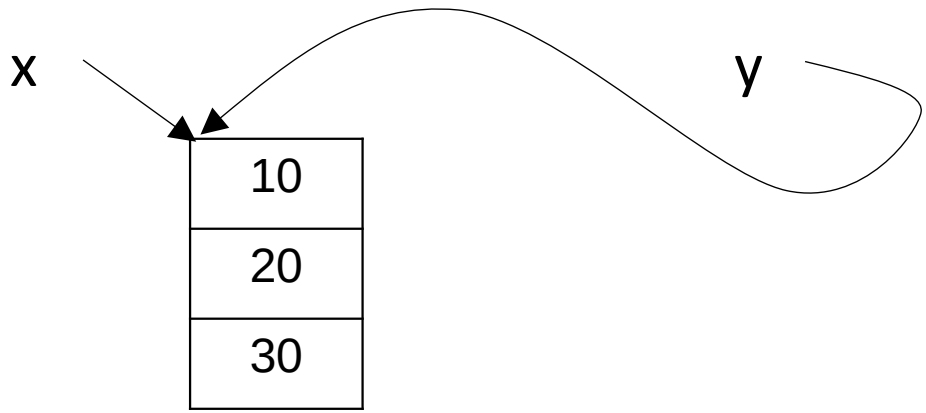


Corrected!

Each assignment statement **copies a reference.**

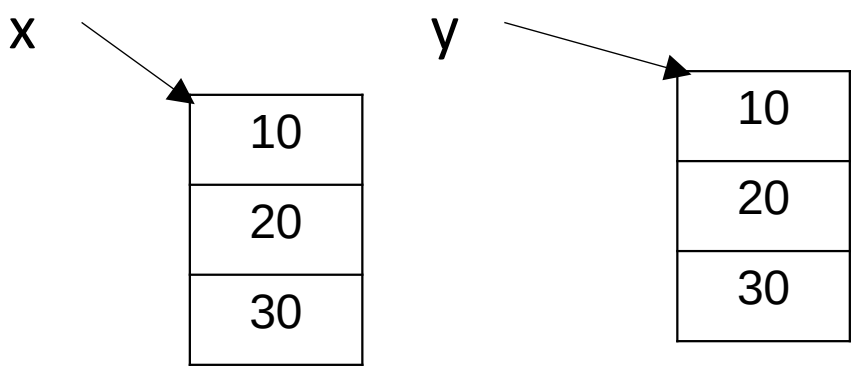
Now, more than one variable point to the same object.

EXERCISE

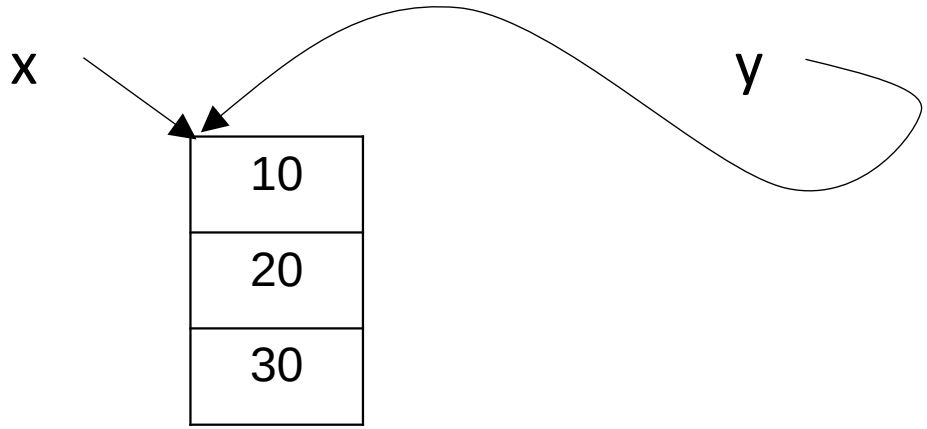


What is the difference between these two diagrams?

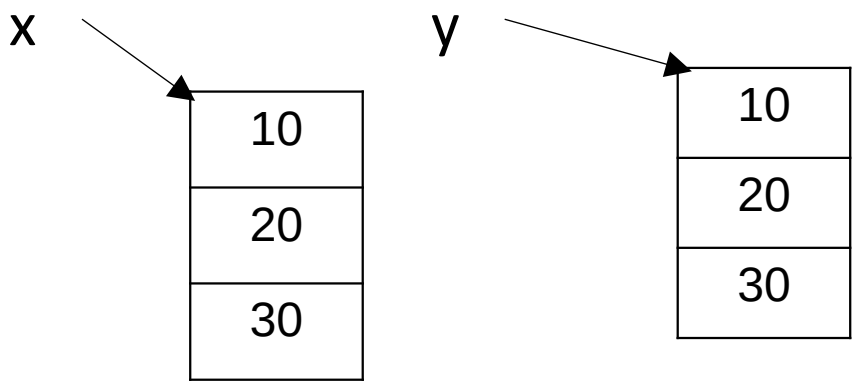
Write some code which will create each one.



EXERCISE



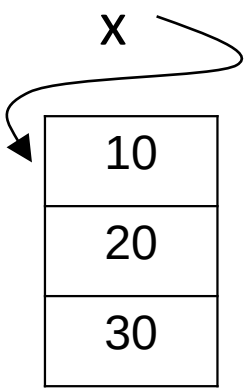
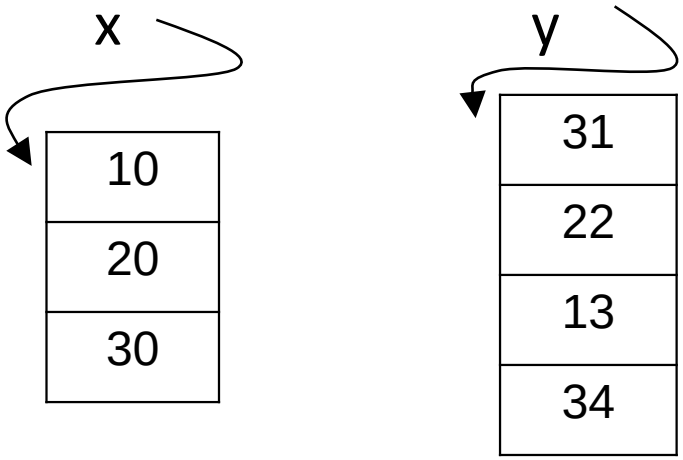

```
x = [10,20,30]
y = x
```



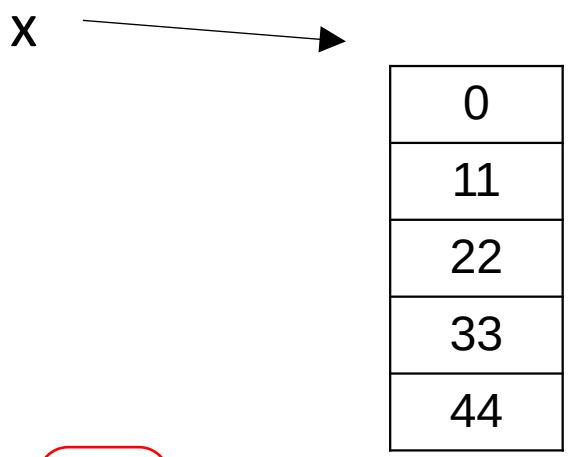

```
x = [10,20,30]
y = [10,20,30]
```

More examples

```
x = [10, 20, 30]
y = [x[2]+1, x[1]+2, x[0]+3, x[-1]+4]
```

Before	After
	

EXERCISE



What lines of code here will result in the diagram above?

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

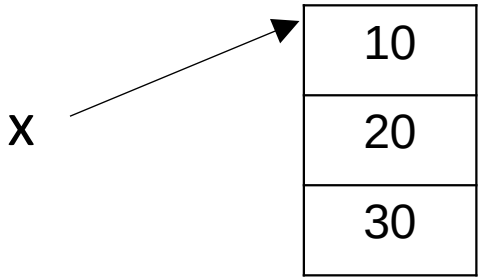
EXERCISE

```
>>> x = [10, 20, 30]
>>> y = ["ab", "cd"]
>>> z = [x, y]
```

← *what is the diagram for z?*

SOLUTION (1 of 3)

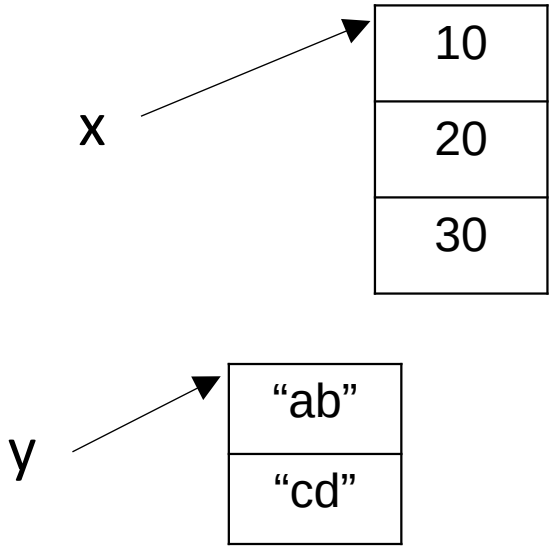
```
>>> x = [10, 20, 30]      # a list containing 3 values
>>> y = ["ab", "cd"]      # a list containing 2 values
>>> z = [x, y]            # a list containing 2 values
```



a list (container)
with 3 values

SOLUTION (2 of 3)

```
>>> x = [10, 20, 30]      # a list containing 3 values
>>> y = ["ab", "cd"]      # a list containing 2 values
>>> z = [x, y]            # a list containing 2 values
```

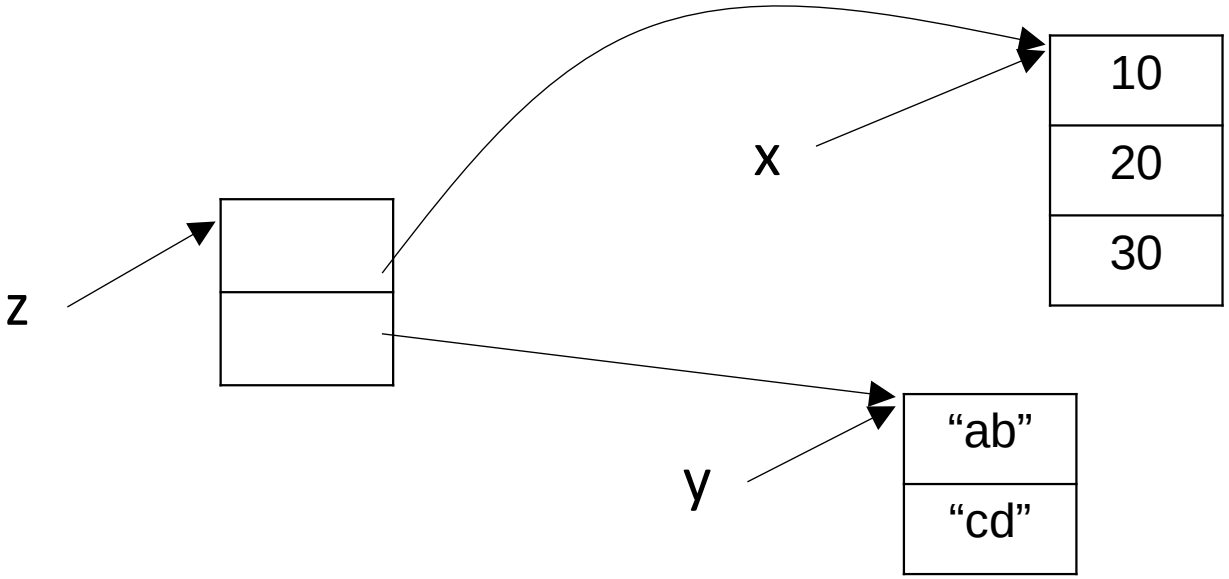


a list (container) with 3 values

a list (container) with 2 values

SOLUTION (3 of 3)

```
>>> x = [10, 20, 30]      # a list containing 3 values
>>> y = ["ab", "cd"]      # a list containing 2 values
>>> z = [x, y]            # a list containing 2 values
```



a list (container) with 3 values

a list (container) with 2 values

EXERCISE

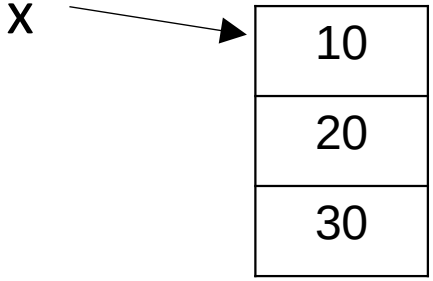
```
>>> x = [10, 20, 30]
>>> y = ["abc", 12, "def", 23]
>>> z = [y, x]

>>> w = [z[0][0], z[1][1]]
```

← *what is the diagram for z?*

← *what is the diagram for w?*

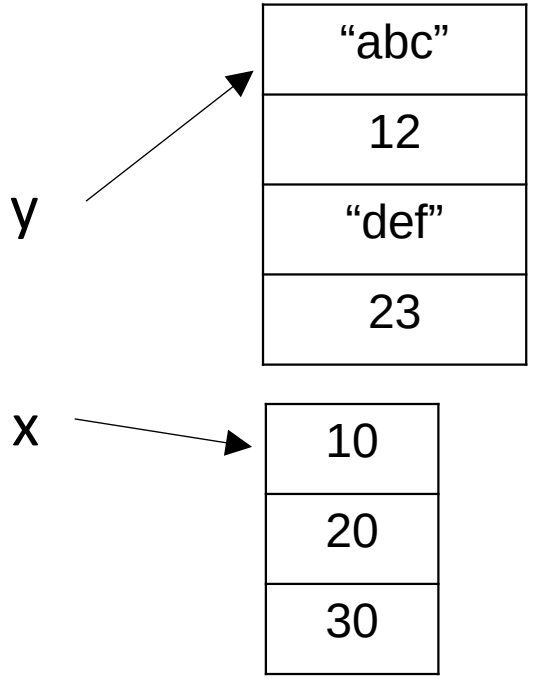
SOLUTION (1 of 4)

```
>>> x = [10, 20, 30]
>>> y = ["abc", 12, "def", 23]
>>> z = [y, x]
>>> w = [ z[0][0], z[1][1] ]
```



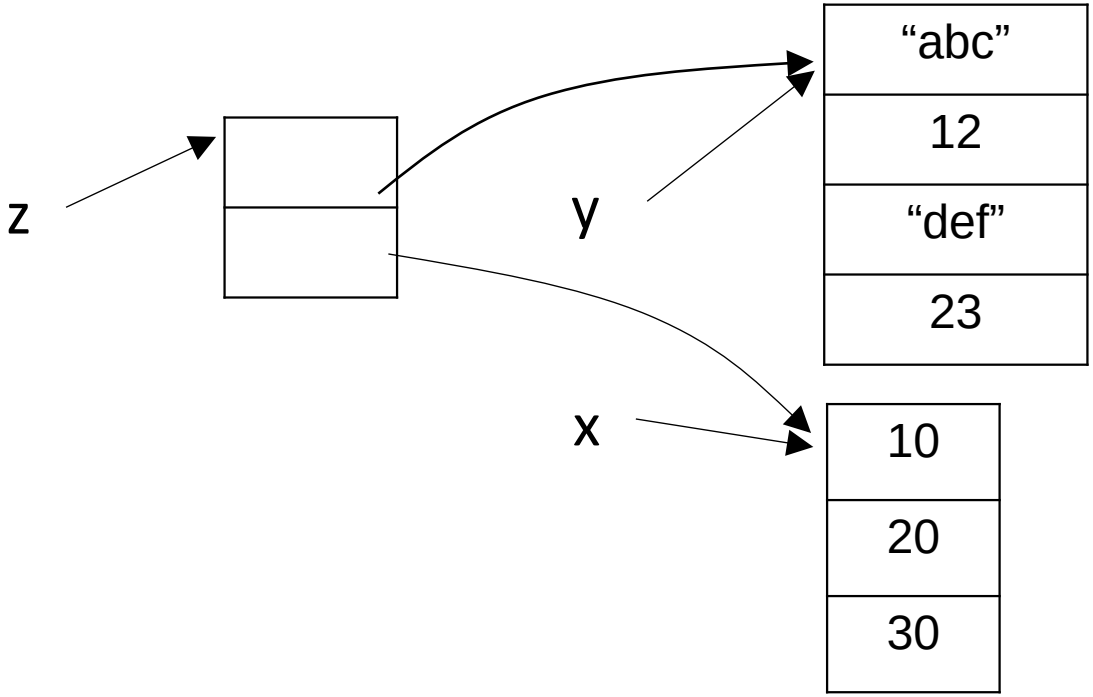
SOLUTION (2 of 4)

```
>>> x = [10, 20, 30]
>>> y = ["abc", 12, "def", 23]
>>> z = [y, x]
>>> w = [ z[0][0], z[1][1] ]
```



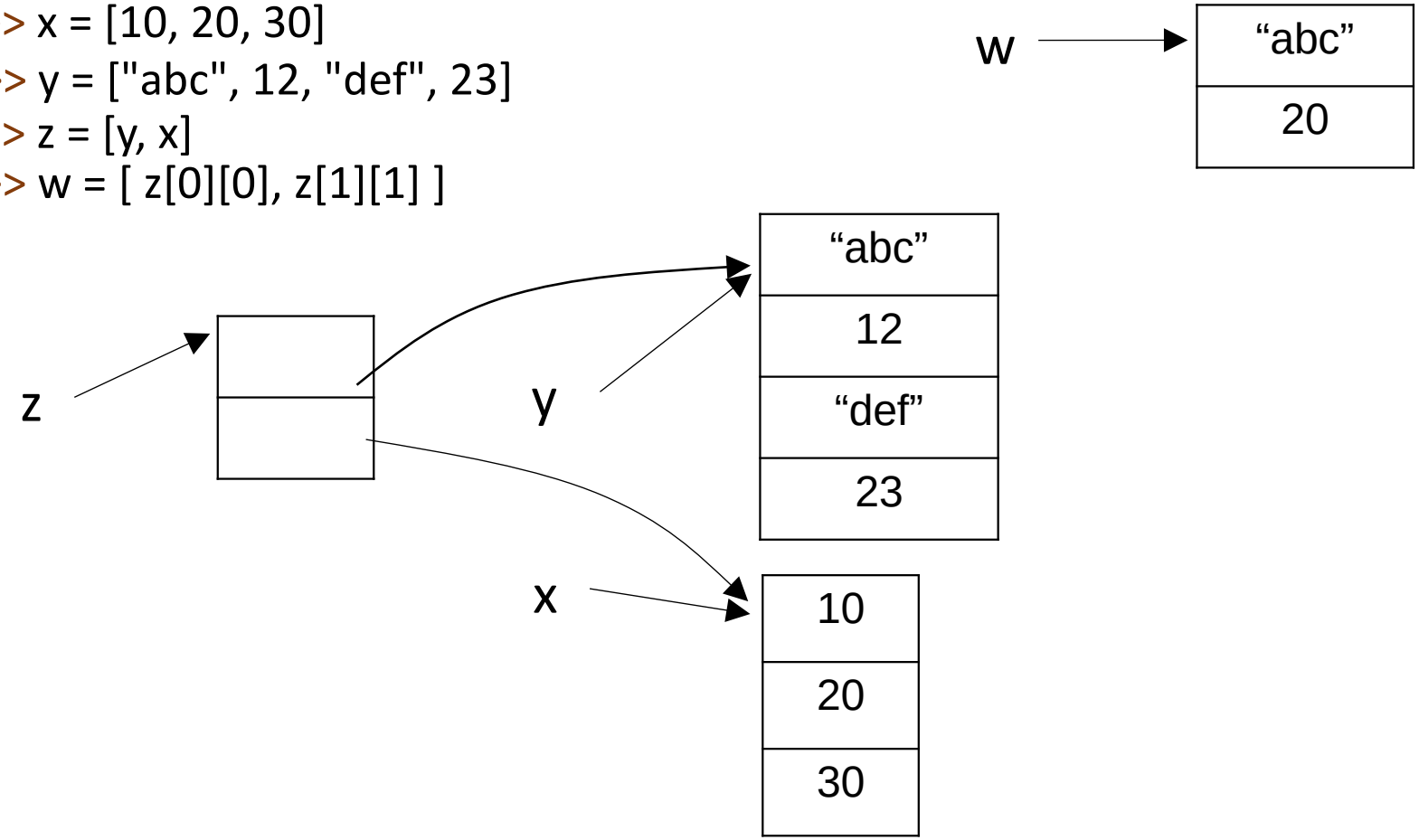
SOLUTION (3 of 4)

```
>>> x = [10, 20, 30]
>>> y = ["abc", 12, "def", 23]
>>> z = [y, x]
>>> w = [ z[0][0], z[1][1] ]
```



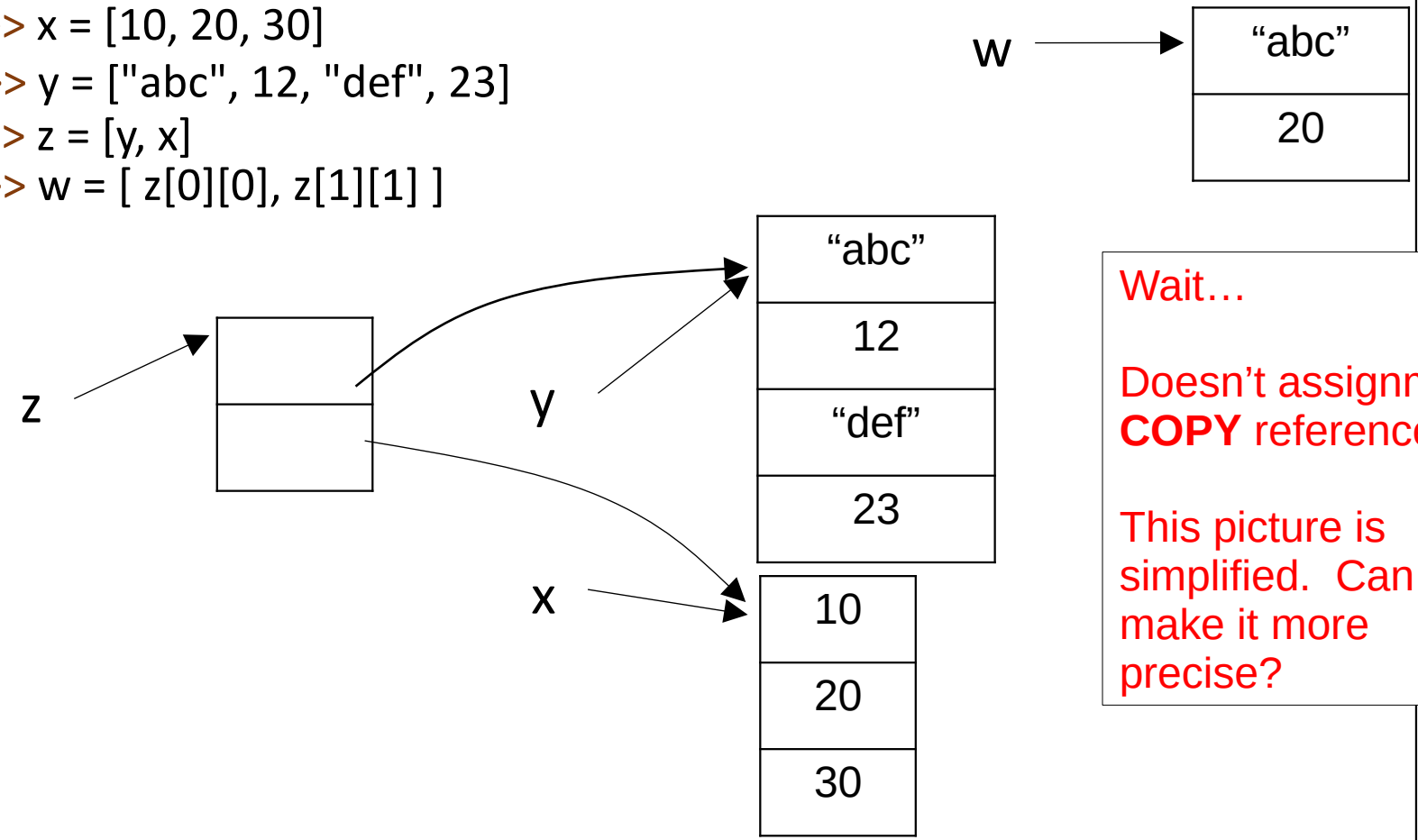
SOLUTION (4 of 4)

```
>>> x = [10, 20, 30]
>>> y = ["abc", 12, "def", 23]
>>> z = [y, x]
>>> w = [ z[0][0], z[1][1] ]
```



SOLUTION

```
>>> x = [10, 20, 30]
>>> y = ["abc", 12, "def", 23]
>>> z = [y, x]
>>> w = [ z[0][0], z[1][1] ]
```



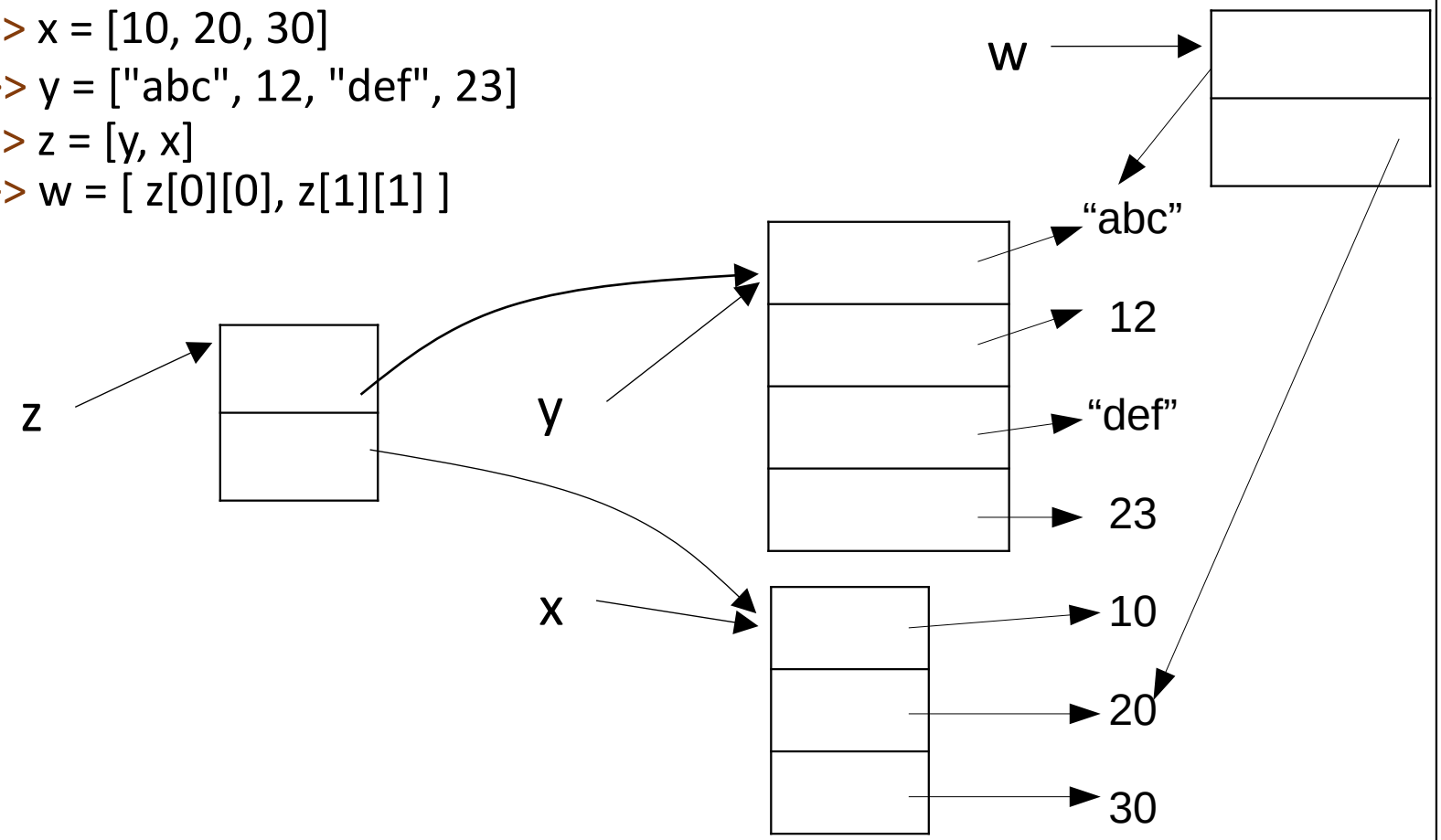
Wait…

Doesn't assignment **COPY** references?

This picture is simplified. Can we make it more precise?

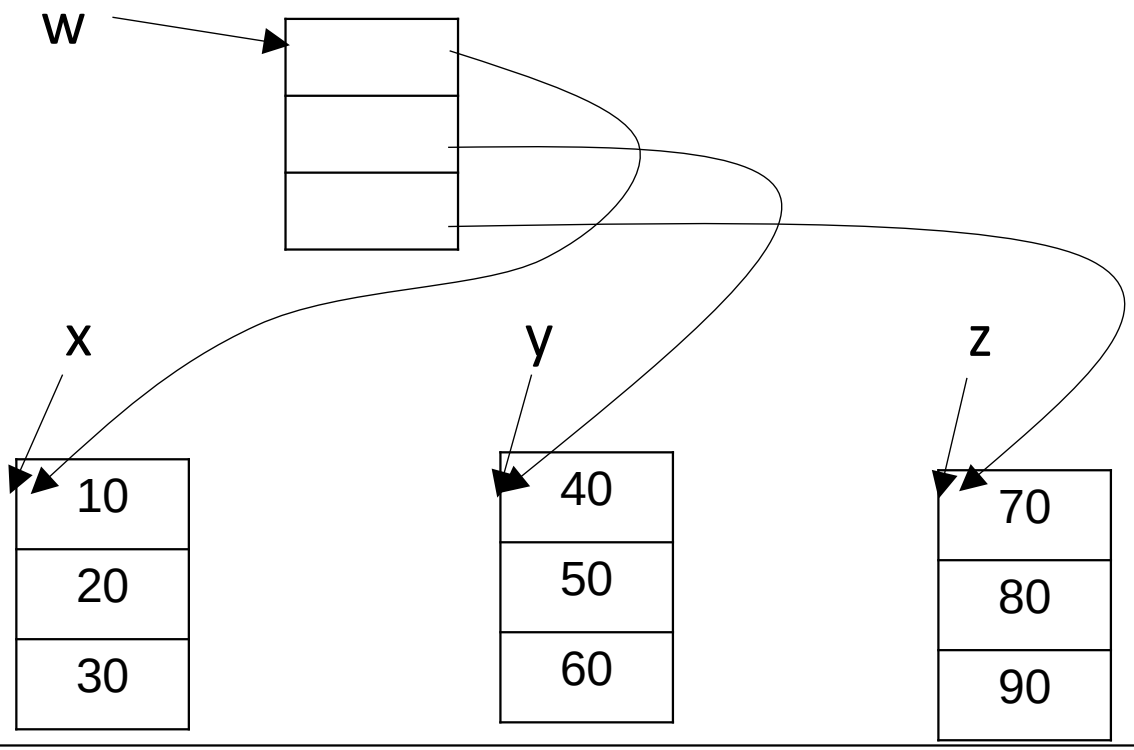
SOLUTION

```
>>> x = [10, 20, 30]
>>> y = ["abc", 12, "def", 23]
>>> z = [y, x]
>>> w = [ z[0][0], z[1][1] ]
```



EXERCISE

What code will produce the following diagram?



Function calls

Arguments and return values

- At a function call $f(arg_1, \ldots, arg_n)$ the called function f is passed references to the values of $arg_1, \ldots, arg_n$

- When a called function returns *val* to the caller, what is returned is a reference to *val*

EXERCISE

```
def myfun(v):
    return [ v[0] ]
```

← *what is the diagram for v?*

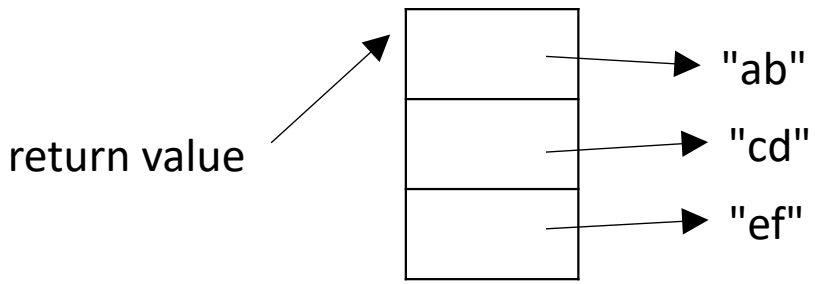
```
x = myfun( [10, 20] )
```

← *what is the diagram for x?*

EXERCISE

Problem:

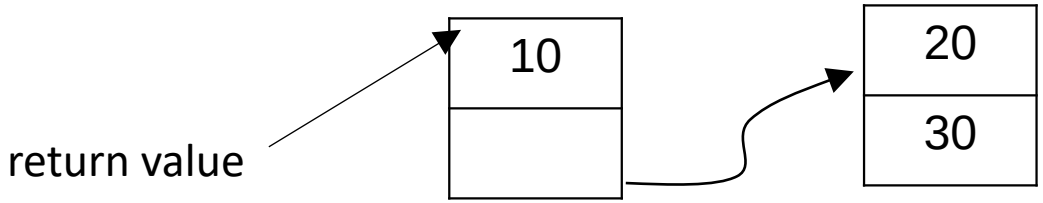
Write a function myfun1() that will return a value whose diagram is:



EXERCISE

Problem:

Write a function myfun2() that will return a value whose diagram is:

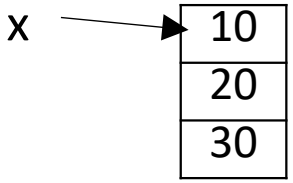


Comparing values

Objects and their values

```
>>> x = [10, 20, 30]
```

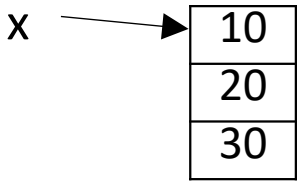
compute RHS value, then store the reference into LHS



Objects and their values

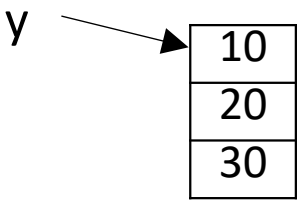
```
>>> x = [10, 20, 30]
```

compute RHS value, then store the reference into LHS




```
>>> y = [10, 20, 30]
```

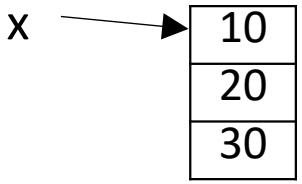
compute RHS value, then store the reference into LHS



Objects and their values

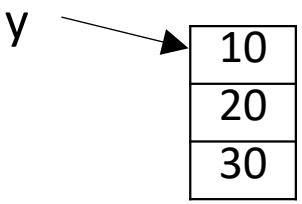
```
>>> x = [10, 20, 30]
```

compute RHS value, then store the reference into LHS




```
>>> y = [10, 20, 30]
```

compute RHS value, then store the reference into LHS



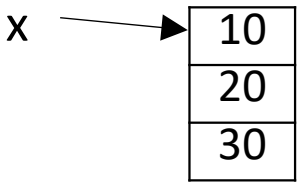
Questions:

- Is id(x) == id(y) ? What do you think? Why?
- Is x == y ? What do you think? Why?

Objects and their values

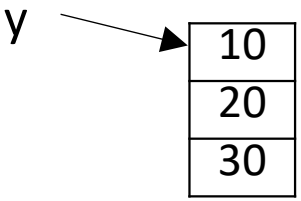
```
>>> x = [10, 20, 30]
```

compute RHS value, then store the reference into LHS




```
>>> y = [10, 20, 30]
```

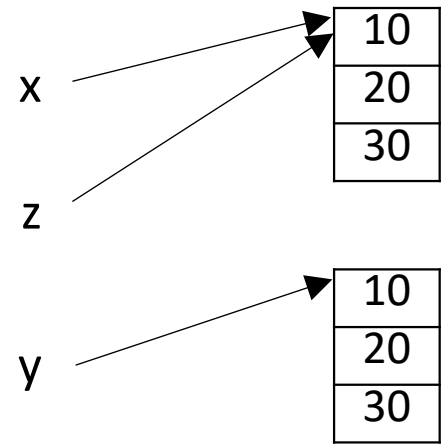
compute RHS value, then store the reference into LHS



- x and y refer to two lists that:
  - have the *same value*
    - same length, same sequence of list elements
  - but are *different objects*
    - they live at different memory locations
    - their id#s are different

is vs. ==

```
>>> x = [10, 20, 30]
>>> y = [10, 20, 30]
>>> z = x
>>> x is z
True
>>> x is y
False
>>> y == z
True
```



a is *b* ≡ "do *a* and *b* refer to the same object?"

a == *b* ≡ "do *a* and *b* have the same value (even if they may refer to different objects)?"

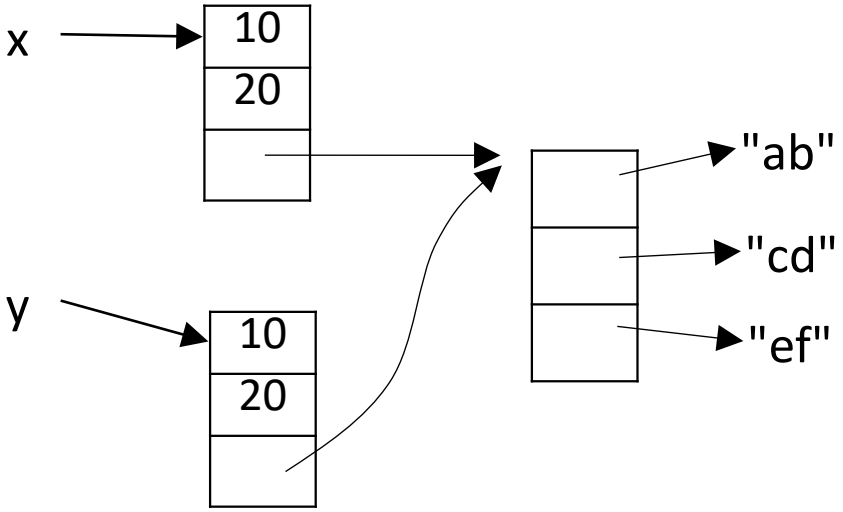
is vs. ==

- If *a* is *b* then *a* == *b*
  - however, *a* == *b* does not necessarily mean *a* is *b*
- In data structure diagrams:
  - *a* is *b* means: *a* and *b* point to the same thing
  - *a* == *b* means: the diagrams for *a* and *b* would match up if they were placed one on top of the other

EXERCISE

True or False:

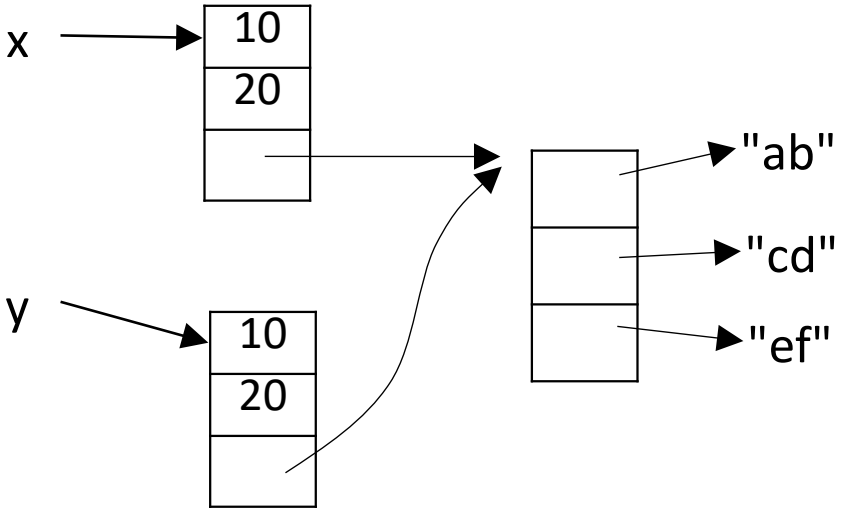
- x is y



EXERCISE

True or False:

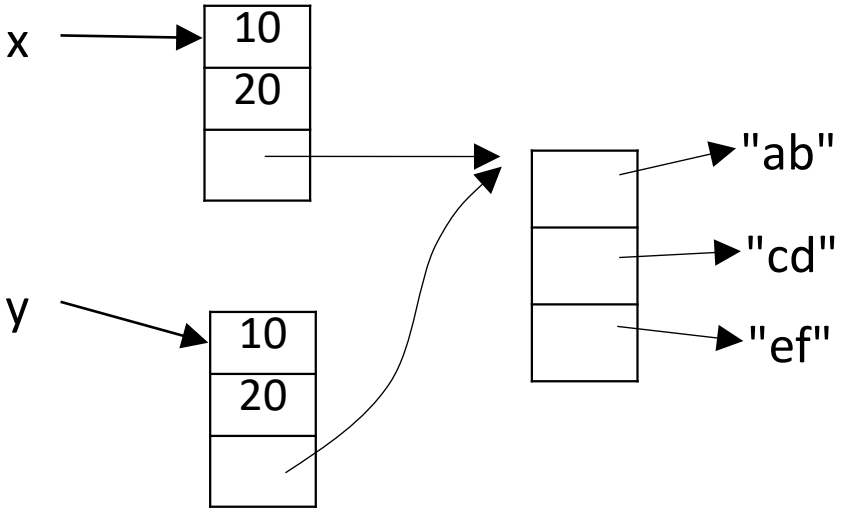
- x is y
- x[2] is y[2]



EXERCISE

True or False:

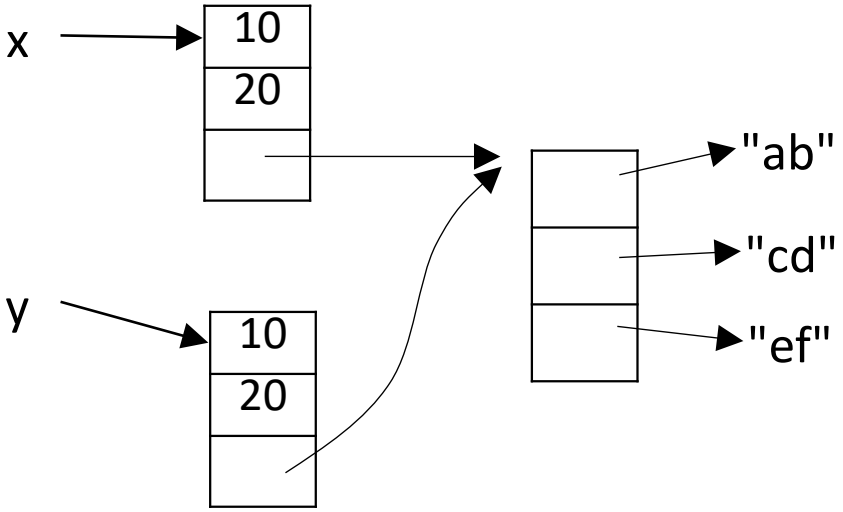
- x is y
- x[2] is y[2]
- x[2][0] is y[2][0]



EXERCISE

True or False:

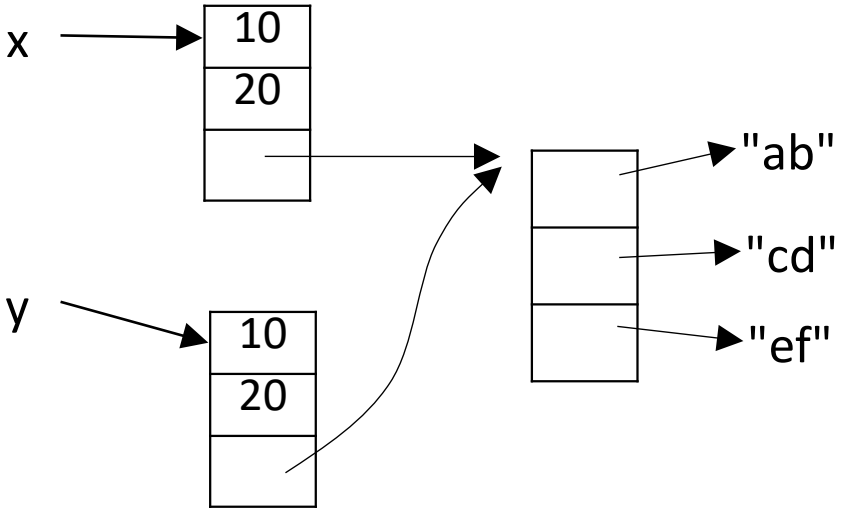
- x is y
- x[2] is y[2]
- x[2][0] is y[2][0]
- x[2] == y[2]



EXERCISE

True or False:

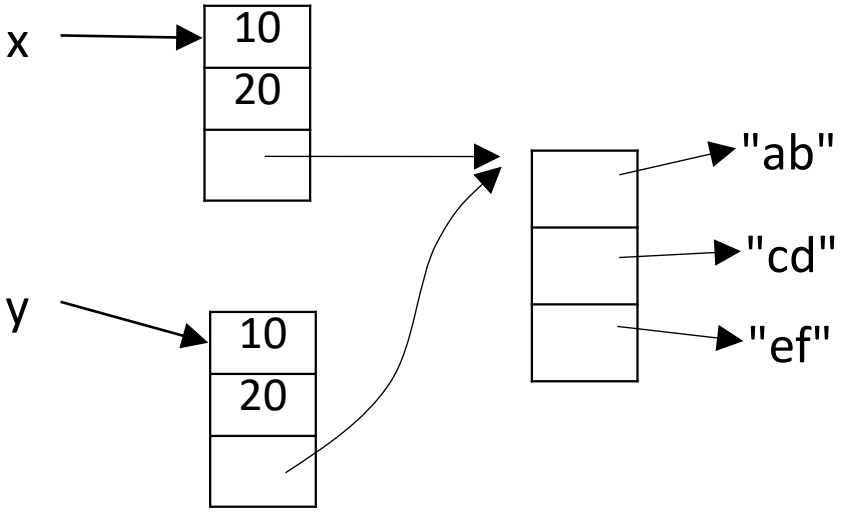
- x is y
- x[2] is y[2]
- x[2][0] is y[2][0]
- x[2] == y[2]
- x[0] == y[0]



EXERCISE

True or False:

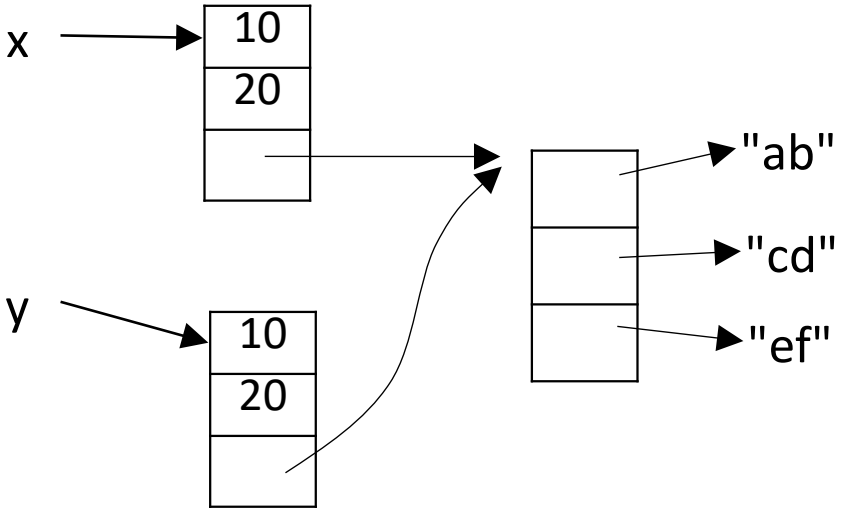
- x is y
- x[2] is y[2]
- x[2][0] is y[2][0]
- x[2] == y[2]
- x[0] == y[0]
- x[0] == y[1]



EXERCISE

True or False:

- x is y
- x[2] is y[2]
- x[2][0] is y[2][0]
- x[2] == y[2]
- x[0] == y[0]
- x[0] == y[1]
- x == y



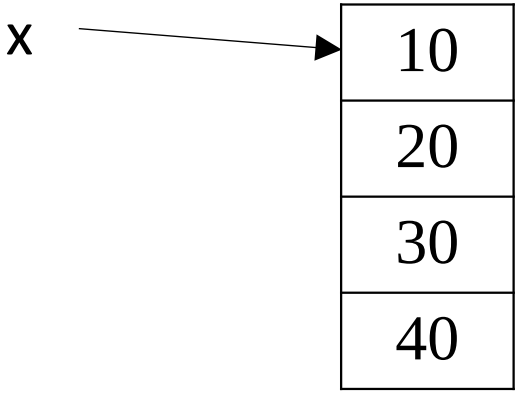
Aliasing

Some simple Python code

```
>>> x = [10, 20, 30, 40]
>>> x
[10, 20, 30, 40]
>>> y = x
>>> y
[10, 20, 30, 40]
>>> x[1] = 999
>>> y
[10, 999, 30, 40]
```

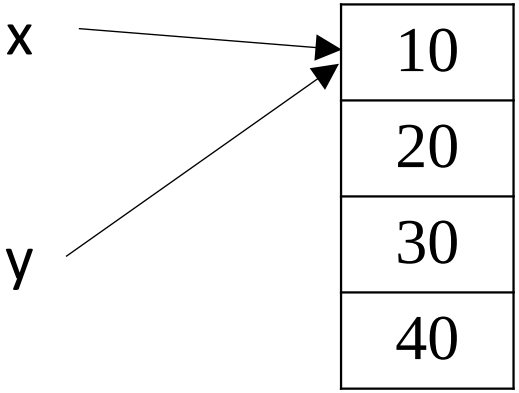
Some simple Python code

```
>>> x = [10, 20, 30, 40]
>>> x
[10, 20, 30, 40]
>>> y = x
>>> y
[10, 20, 30, 40]
>>> x[1] = 999
>>> y
[10, 999, 30, 40]
```



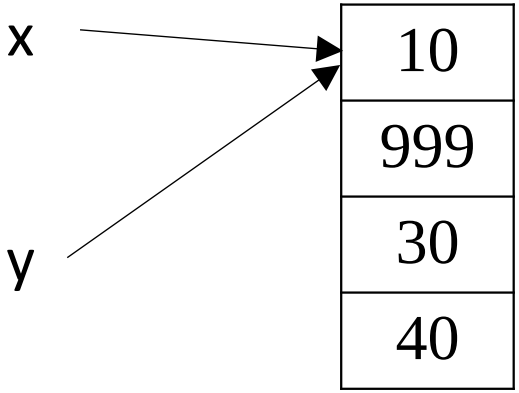
Some simple Python code

```
>>> x = [10, 20, 30, 40]
>>> x
[10, 20, 30, 40]
>>> y = x
>>> y
[10, 20, 30, 40]
>>> x[1] = 999
>>> y
[10, 999, 30, 40]
```



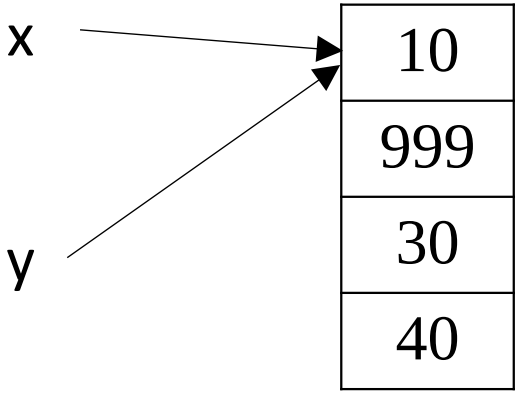
Some simple Python code

```
>>> x = [10, 20, 30, 40]
>>> x
[10, 20, 30, 40]
>>> y = x
>>> y
[10, 20, 30, 40]
>>> x[1] = 999
>>> y
[10, 999, 30, 40]
```



Some simple Python code

```
>>> x = [10, 20, 30, 40]
>>> x
[10, 20, 30, 40]
>>> y = x
>>> y
[10, 20, 30, 40]
>>> x[1] = 999
>>> y
[10, 999, 30, 40]
```



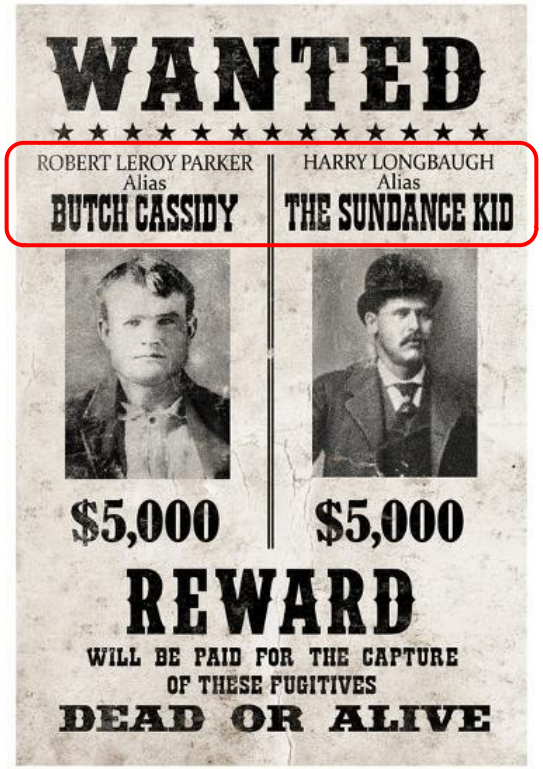
y refers to the same data structure as x

Aliasing

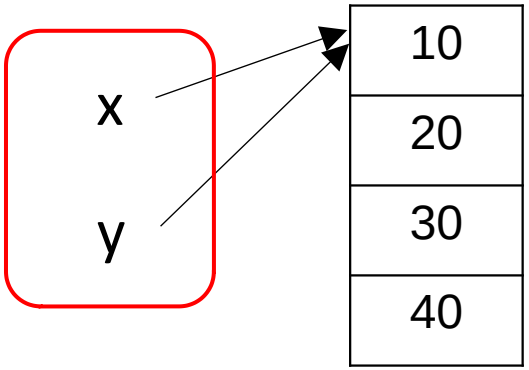
Aliasing refers to the situation where there are multiple different references to (≈ names for) the same value

- the different references are said to be *aliases* of each other

aliases



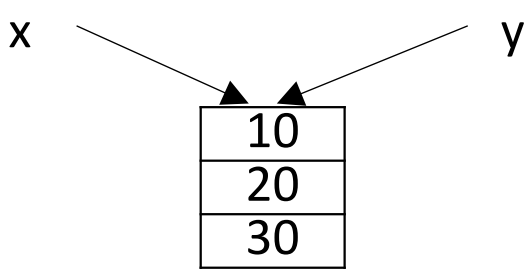
aliases

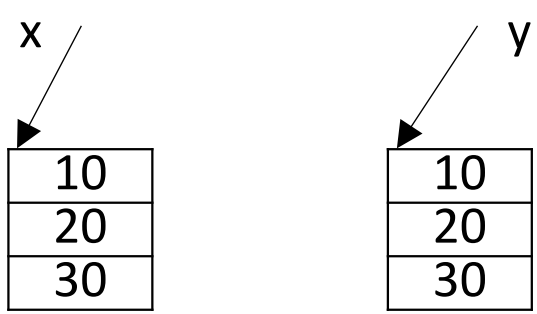


Creating aliases

Aliasing occurs when we create *multiple copies of a reference* to some value

Aliasing	No aliasing
x = [10, 20, 30]	x = [10, 20, 30]
y = x	y = [10, 20, 30]





EXERCISE

```
x = [10, 20, 30]
y = [x, x]
```

← *what is the diagram for y?*

based on the diagram, what can you say about aliasing in y?

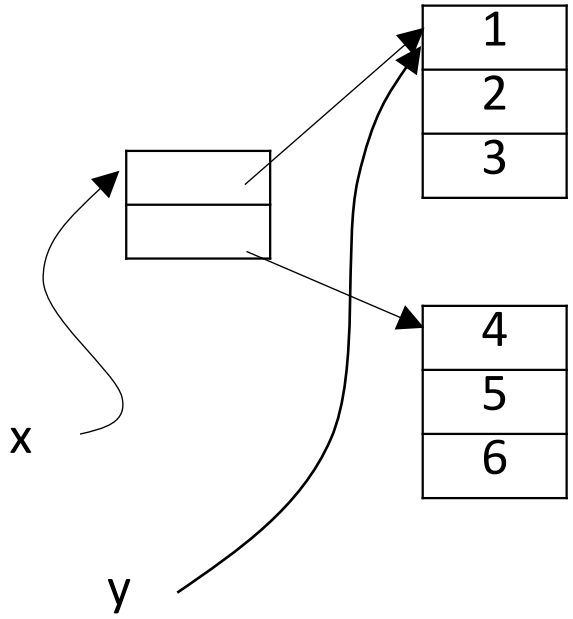
EXERCISE

```
>>> def foo(w):
        return w[0]

>>> x = [[1, 2, 3], [4, 5, 6]]
>>> y = foo(x)

>>> y[1] = 999
>>> x
```

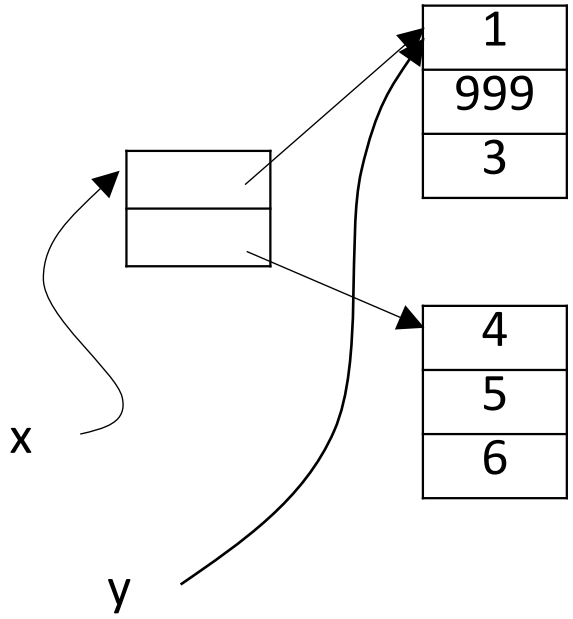
← ① *what is the diagram for x and y?*

← ② *what do you think will be printed out?*

SOLUTION



The diagram at ①



The diagram at ②: the value printed is: [[1,999,3], [4,5,6]]

Detecting aliasing

If:

- you change the value of one variable (or data structure); and
- the value of some other variable (or data structure) changes in the same way

this is likely to be due to aliasing

Example

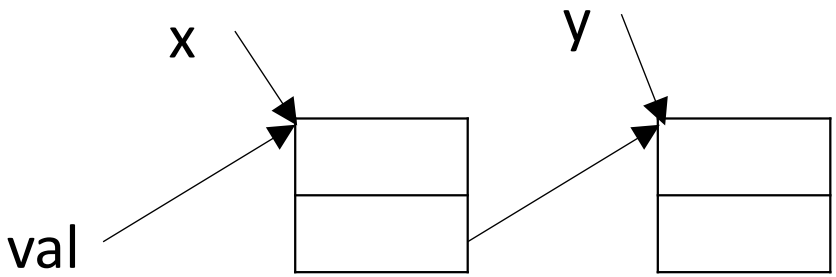
```
>>> def foo(w):
    return w[0]

>>> x = [[1,2,3], [4,5,6]]
>>> y = foo(x)
>>> y[1] = 55
>>> x
[[1, 55, 3], [4, 5, 6]]
```

EXERCISE

Problem:

Write a function myfun2(x, y) that will return a value whose diagram is shown below. You can assume that both x and y are lists of length 2.

```
def myfun2(x, y):
    …

    val = …
    # diagram for val here
    return val
```

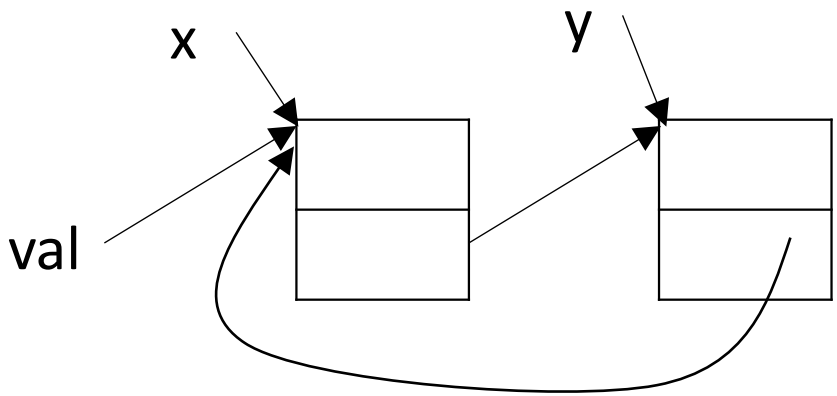


EXERCISE

Problem:

Write a function myfun3(x, y) that will return a value whose diagram is shown below. You can assume that both x and y are lists of length 2.

```
def myfun3(x, y):
    …

    val = …
    # diagram for val here
    return val
```

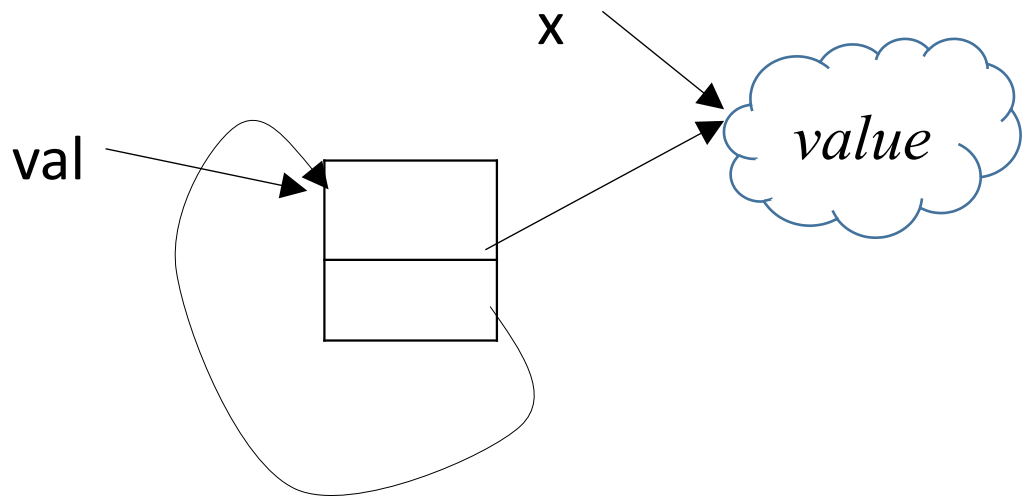


EXERCISE

Problem:

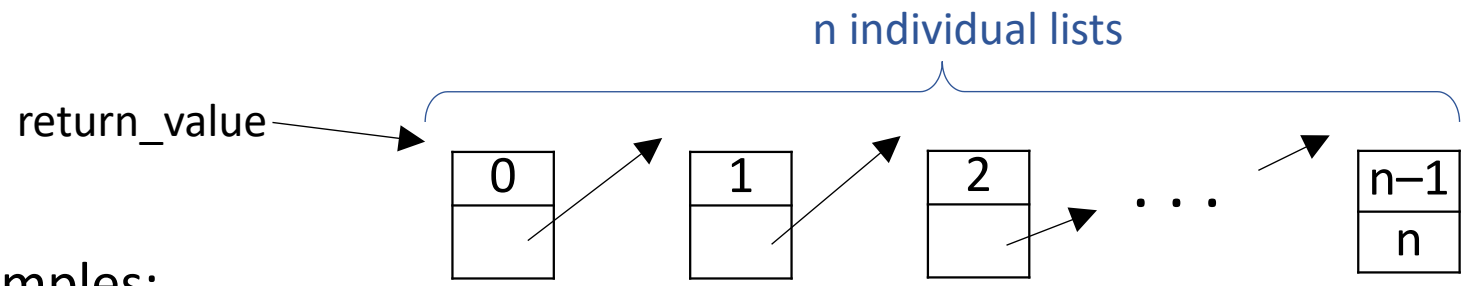
Write a function myfun4(x) that will return a value whose diagram is shown below.

```
def myfun4(x):
    …

    val = …
    # diagram for val here
    return val
```



EXERCISE

Define a function chain(n), where n ≥ 0 is an integer, that returns a value whose diagram looks like this:



Examples:

value of n	0	2	5
return value	[ ]