SISE

הנדסת מערכות תוכנה ומידע

Software and Information Systems Engineering

Introduction to computer science in Python

Fall 2021-22

Department of Software and Information Systems Engineering

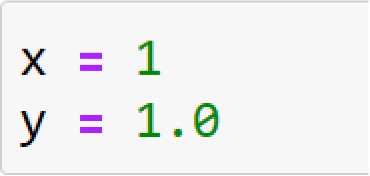
Ben-Gurion University of the Negev

Topic 3: Memory model

Python’s memory model

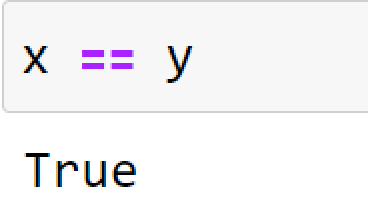
- **Equality and identity**
- **Assignments: mutable vs. immutable types**
- Passing arguments to functions
- The function stack
- Local versus global variables

Equality ≠ Identity in Python



```
x = 1
y = 1.0
```

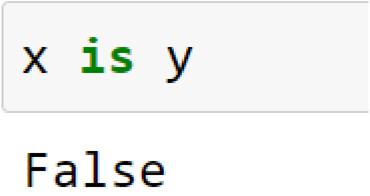
- Equality of an integer and a float: cast int to float, check equality of the two values



```
x == y
```

```
True
```

- Are these two objects (numbers) identical?



```
x is y
```

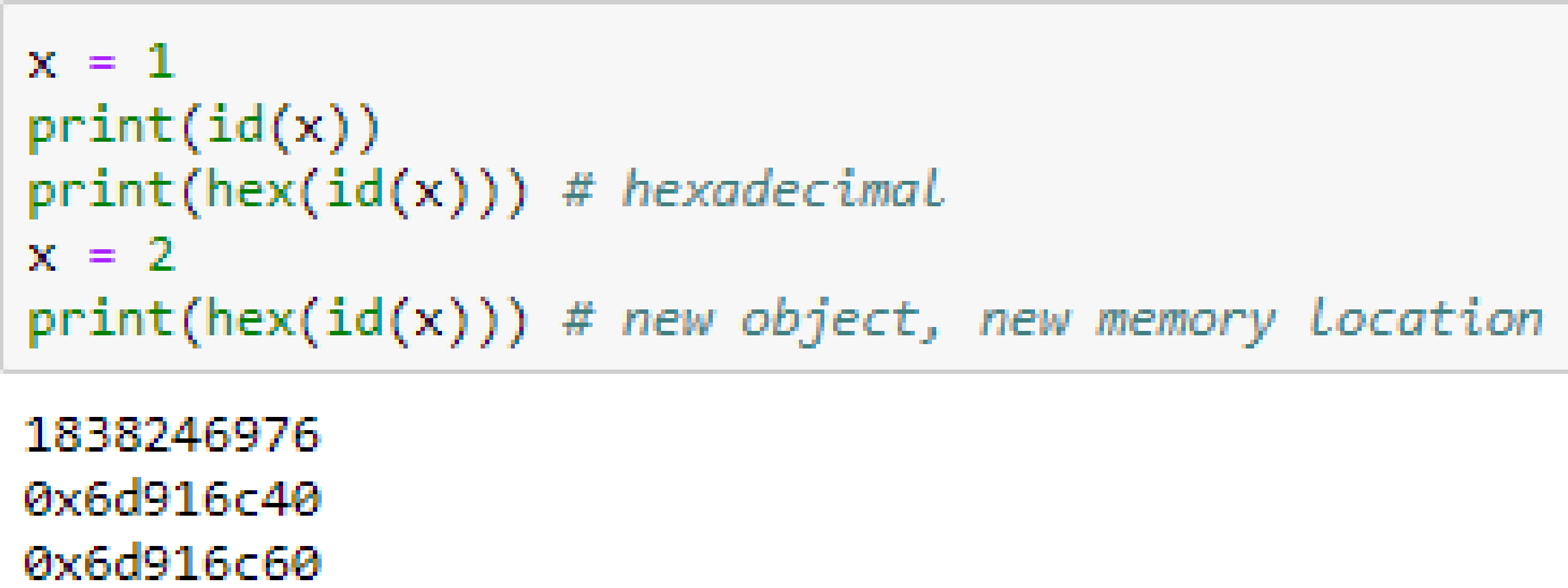
```
False
```

- These identity operators is and is not examine if the two objects referred to are the same object in memory. Identity is “stricter” than equality: identity → equality, but **not** vice versa

Python's *id* Function

Returns an integer which is guaranteed to be unique and constant for this object during its lifetime → the address of the object in memory

id(*object1*) == id(*object2*) if and only if *object1* is *object2*

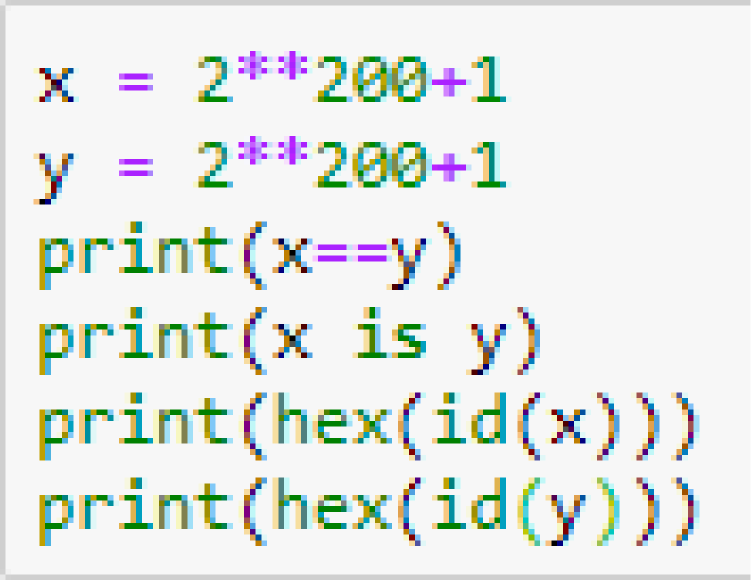


```
x = 1
print(id(x))
print(hex(id(x))) # hexadecimal
x = 2
print(hex(id(x))) # new object, new memory location
```

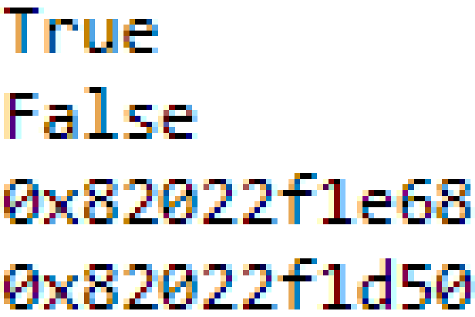
```
1838246976
0x6d916c40
0x6d916c60
```

Python’s memory model

The address of an object is typically not uniquely determined by its value



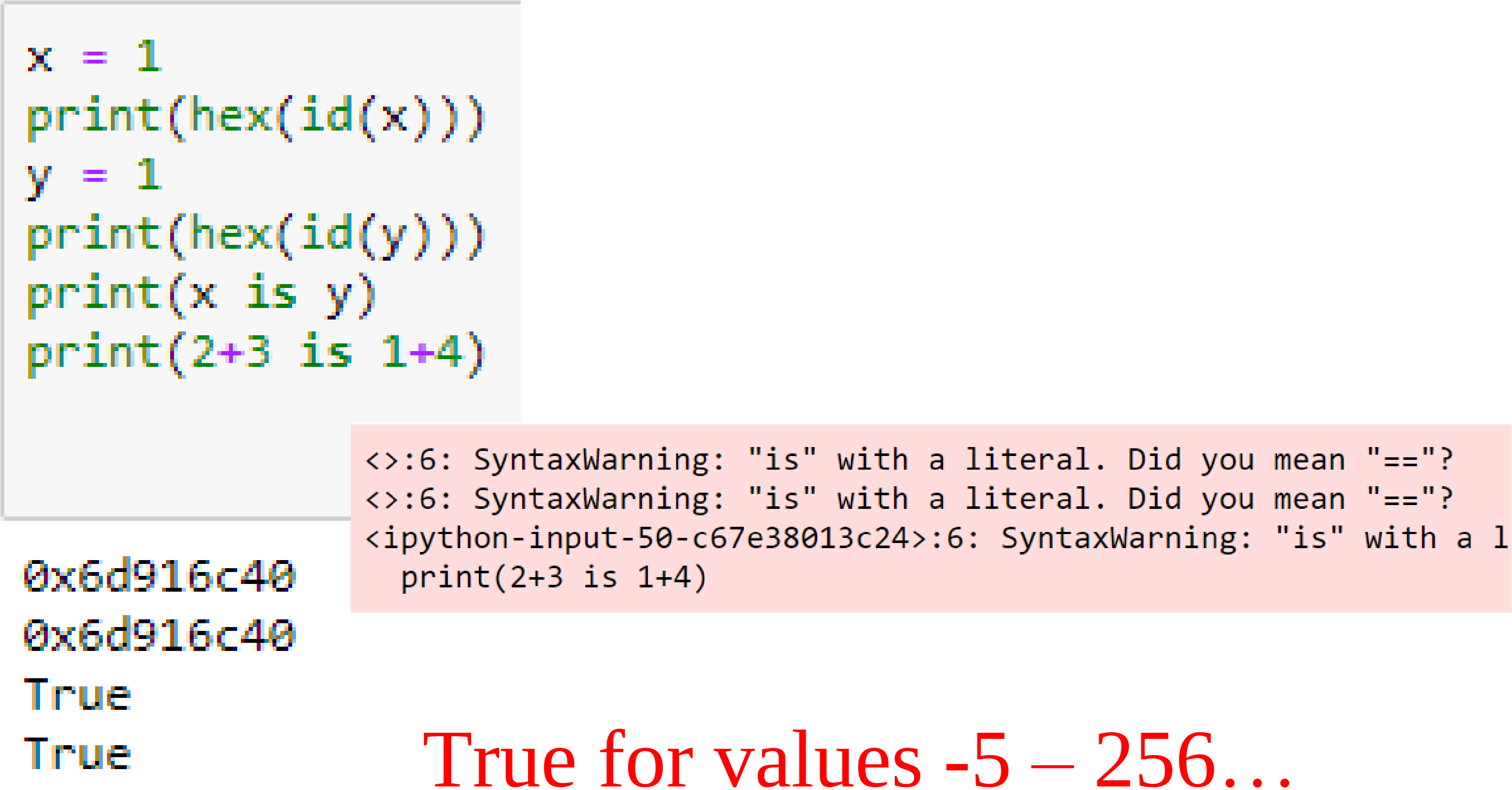
```
x = 2**200+1
y = 2**200+1
print(x==y)
print(x is y)
print(hex(id(x)))
print(hex(id(y)))
```



```
True
False
0x82022f1e68
0x82022f1d50
```

Constant memory address for “small“ immutable objects

- Goal: optimize memory access
- Constant memory address independent of execution history



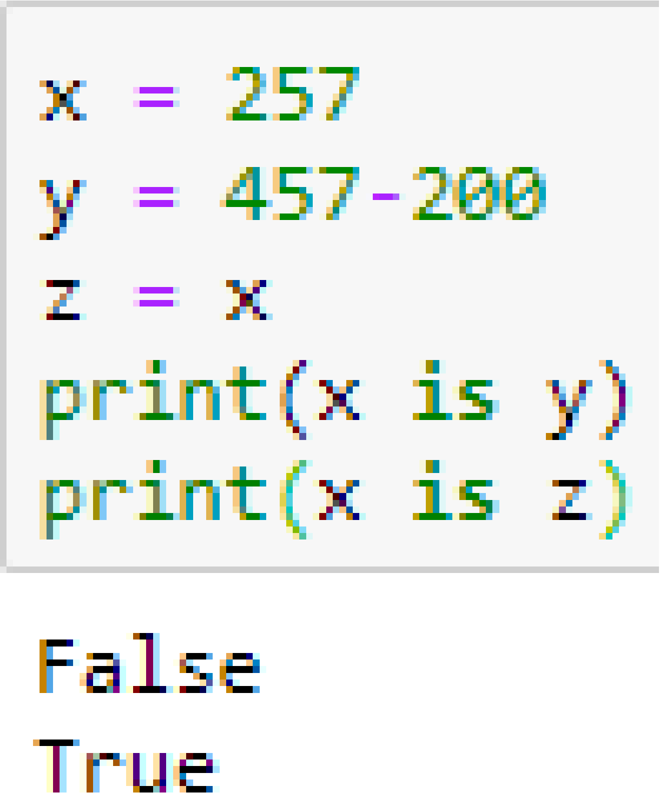
```
x = 1
print(hex(id(x)))
y = 1
print(hex(id(y)))
print(x is y)
print(2+3 is 1+4)
```

```
<>:6: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:6: SyntaxWarning: "is" with a literal. Did you mean "=="?
<ipython-input-50-c67e38013c24>:6: SyntaxWarning: "is" with a l
  print(2+3 is 1+4)
```

```
0x6d916c40
0x6d916c40
True
True
```

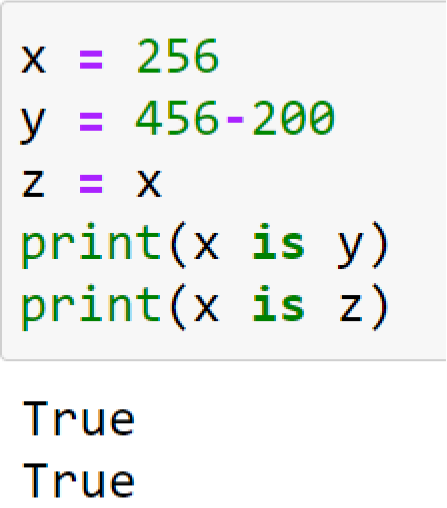
True for values -5 – 256…

The effect of assignment



```python
x = 257
y = 457-200
z = x
print(x is y)
print(x is z)
```

```
False
True
```



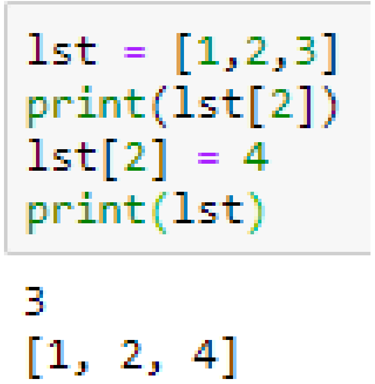
```python
x = 256
y = 456-200
z = x
print(x is y)
print(x is z)
```

```
True
True
```

Upon z = x : the variable z now refers to the same object as x, no new object is created!

Mutable and immutable objects

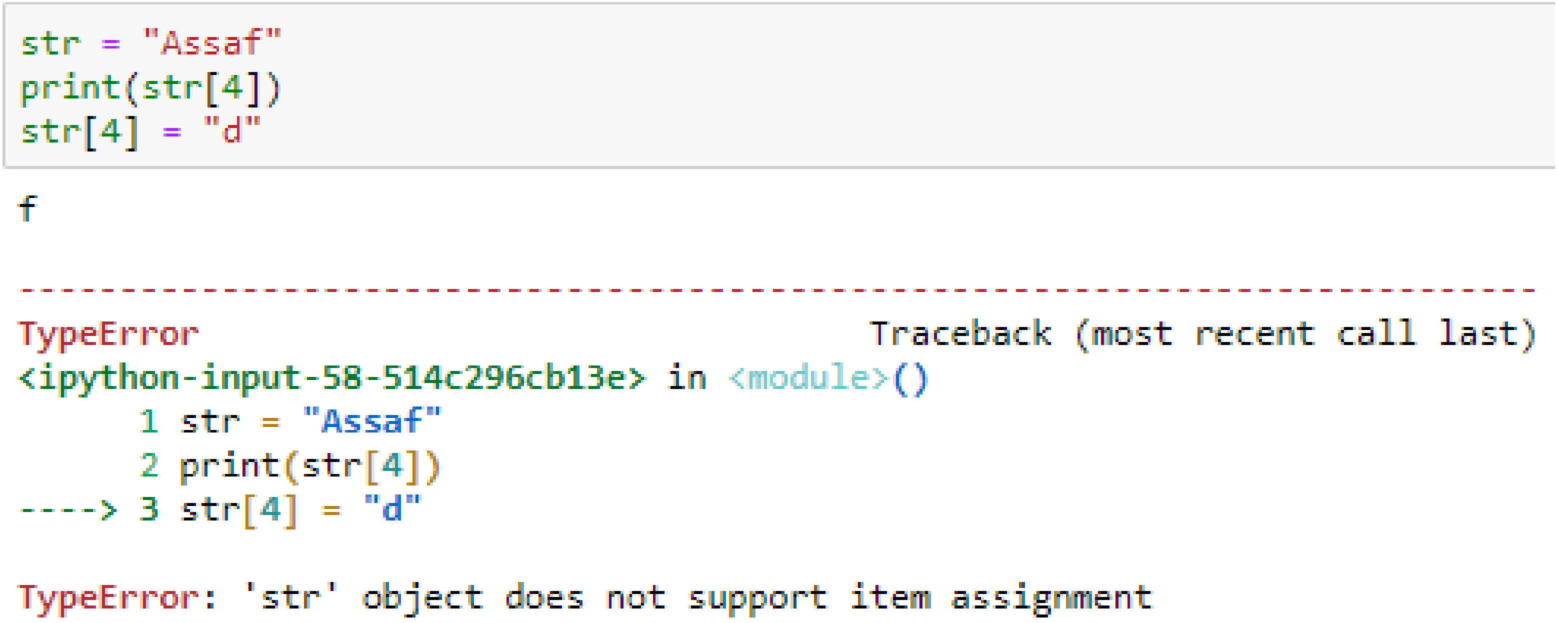
- Lists are mutable objects



```
lst = [1,2,3]
print(lst[2])
lst[2] = 4
print(lst)
```

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

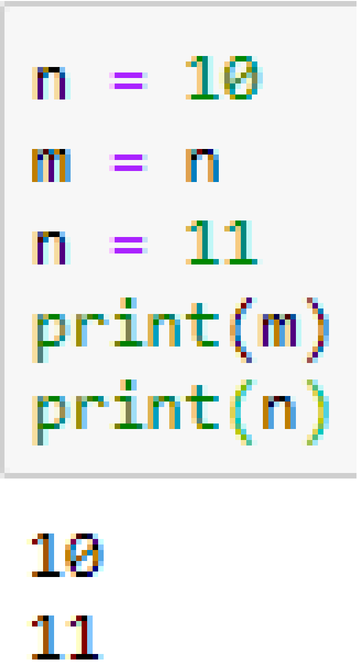
- Strings are **not** mutable (immutable)



```
str = "Assaf"
print(str[4])
str[4] = "d"
```

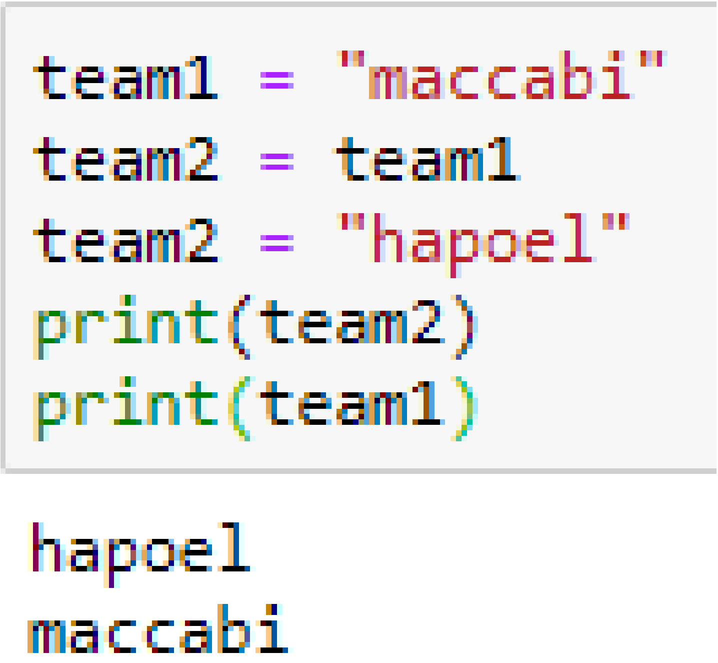
```
f

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-58-514c296cb13e> in <module>()
      1 str = "Assaf"
      2 print(str[4])
----> 3 str[4] = "d"

TypeError: 'str' object does not support item assignment
```

Assignment and reassignment (no surprises)



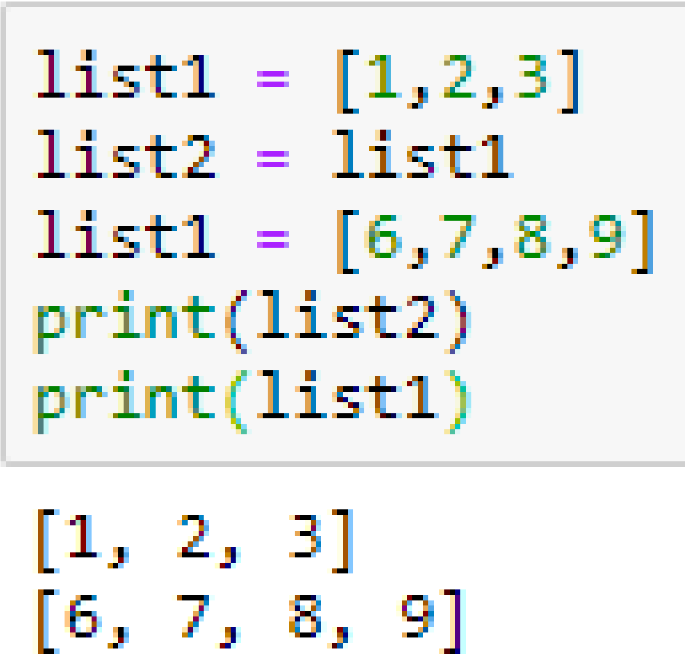
```
n = 10
m = n
n = 11
print(m)
print(n)
```

```
10
11
```



```
team1 = "maccabi"
team2 = team1
team2 = "hapoel"
print(team2)
print(team1)
```

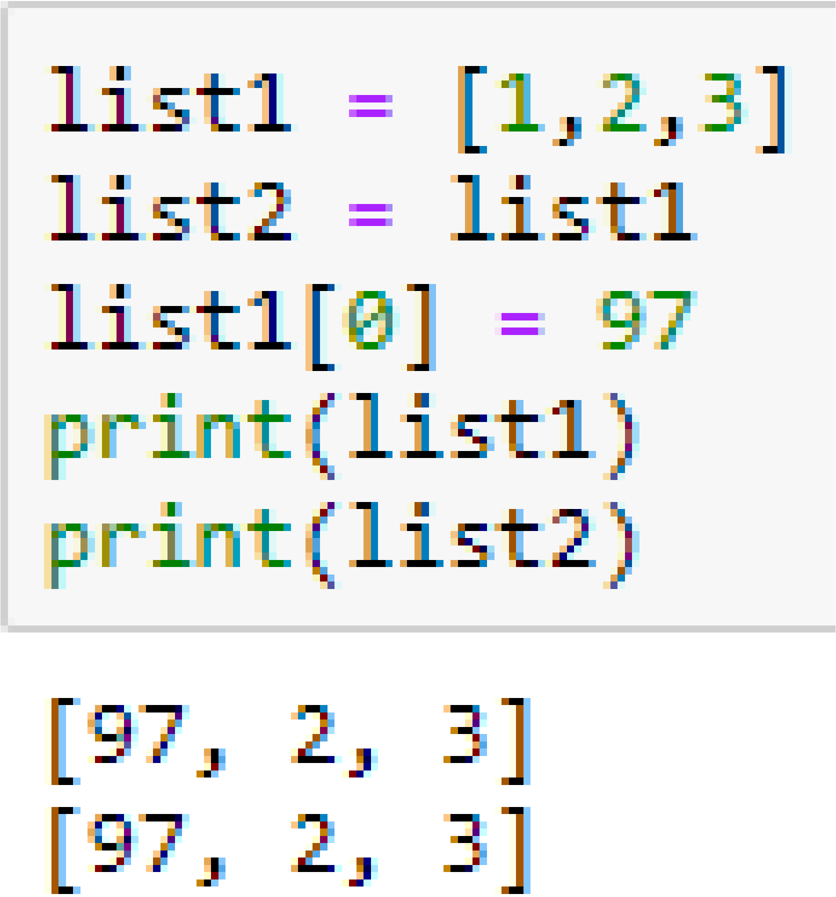
```
hapoel
maccabi
```



```
list1 = [1,2,3]
list2 = list1
list1 = [6,7,8,9]
print(list2)
print(list1)
```

```
[1, 2, 3]
[6, 7, 8, 9]
```

Assignment and mutation



```python
list1 = [1,2,3]
list2 = list1
list1[0] = 97
print(list1)
print(list2)
```

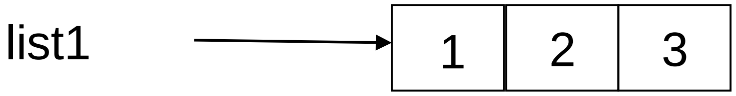
```
[97, 2, 3]
[97, 2, 3]
```

Assignments vs. mutation

```
>>> list1 = [1,2,3]

>>> list2 = list1

>>> list1[0] = 97
```

Assignments vs. mutation

```
>>> list1 = [1,2,3]
>>> list2 = list1
>>> list1[0] = 97
```



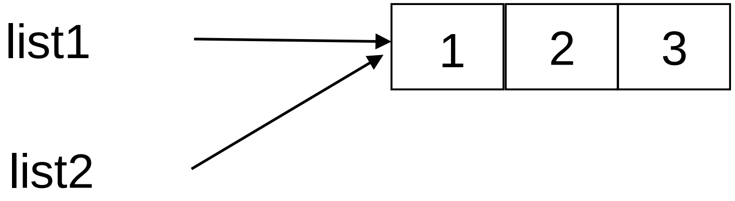
list1

1 2 3

- The assignment list1 = [1,2,3] creates a list object, [1,2,3], and a reference from the variable name, list1, to this object.

Assignments vs. mutation

```
>>> list1 = [1,2,3]

>>> list2 = list1

>>> list1[0] = 97
```



list1

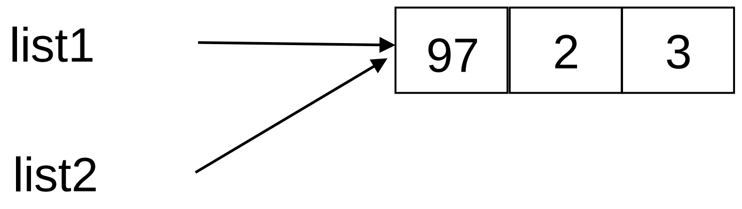
list2

1 2 3

- The assignment list1 = [1,2,3] creates a list object, [1,2,3], and a reference from the variable name, list1, to this object.
- The assignment list2 = list1 does not create a new object. It just creates a new variable name, list2, which now refers to the same object

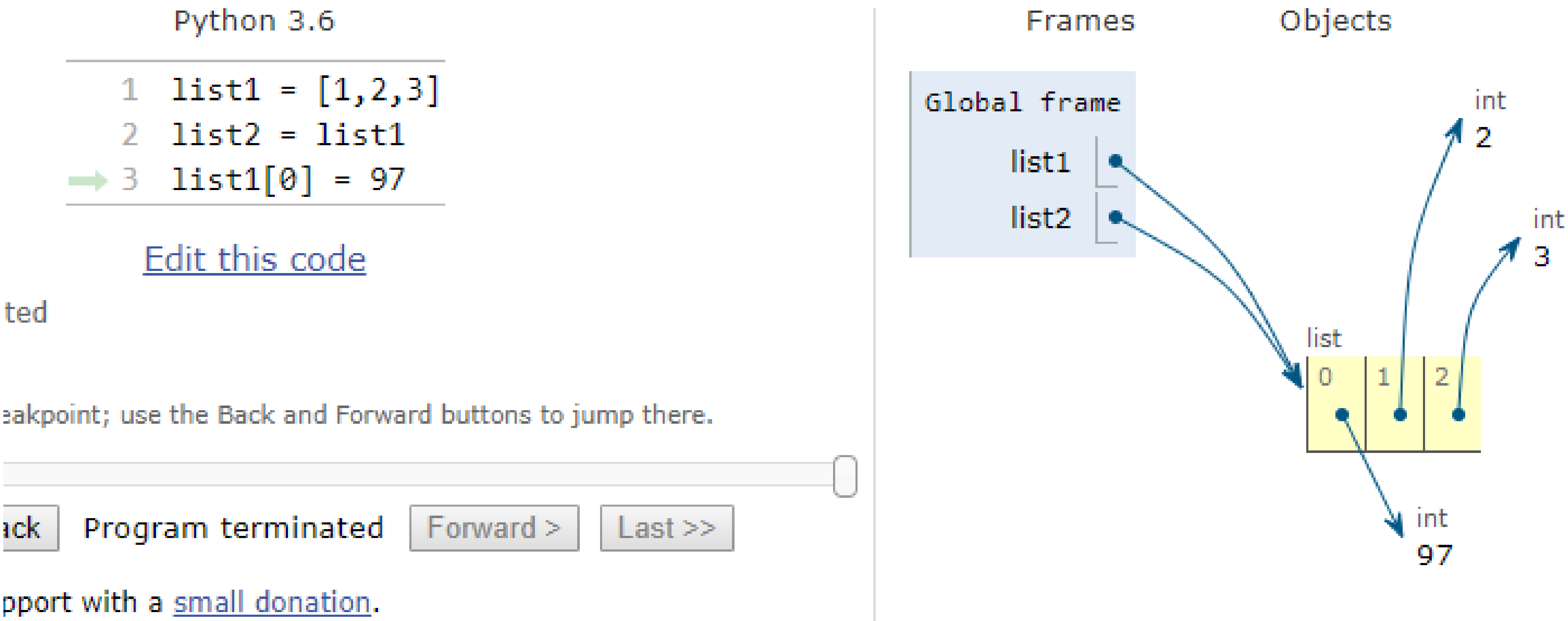
Assignments vs. mutation

```
>>> list1 = [1,2,3]

>>> list2 = list1

>>> list1[0] = 97
```



- The assignment list1 = [1,2,3] creates a list object, [1,2,3], and a reference from the variable name, list1, to this object.
- The assignment list2 = list1 does not create a new object. It just creates a new variable name, list2, which now refers to the same object.
- When we mutate list1[0] = 97, we do not change these references. Thus, displaying list2 produces [97,2,3].

A graphical view: Python Tutor



Python 3.6

```
list1 = [1,2,3]
list2 = list1
list1[0] = 97
```

Edit this code

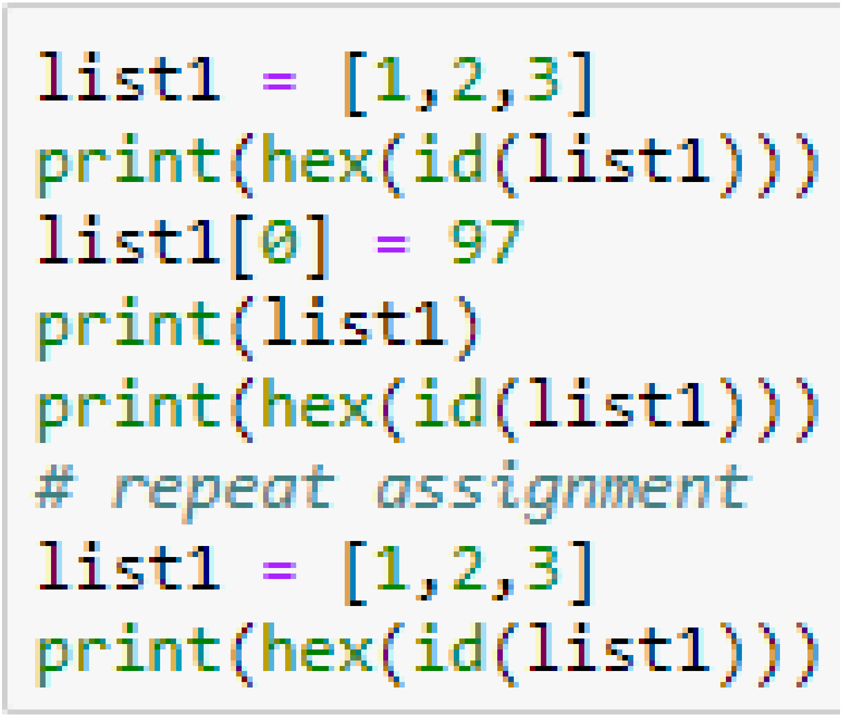
ted

eakpoint; use the Back and Forward buttons to jump there.

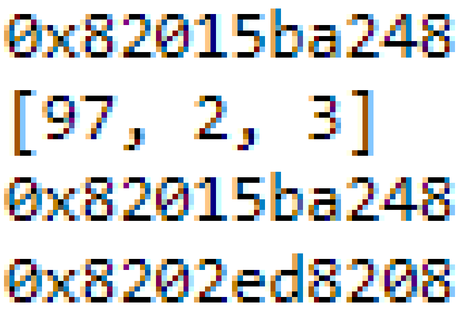
Program terminated

pport with a small donation.

Mutation does not change the memory location of an object

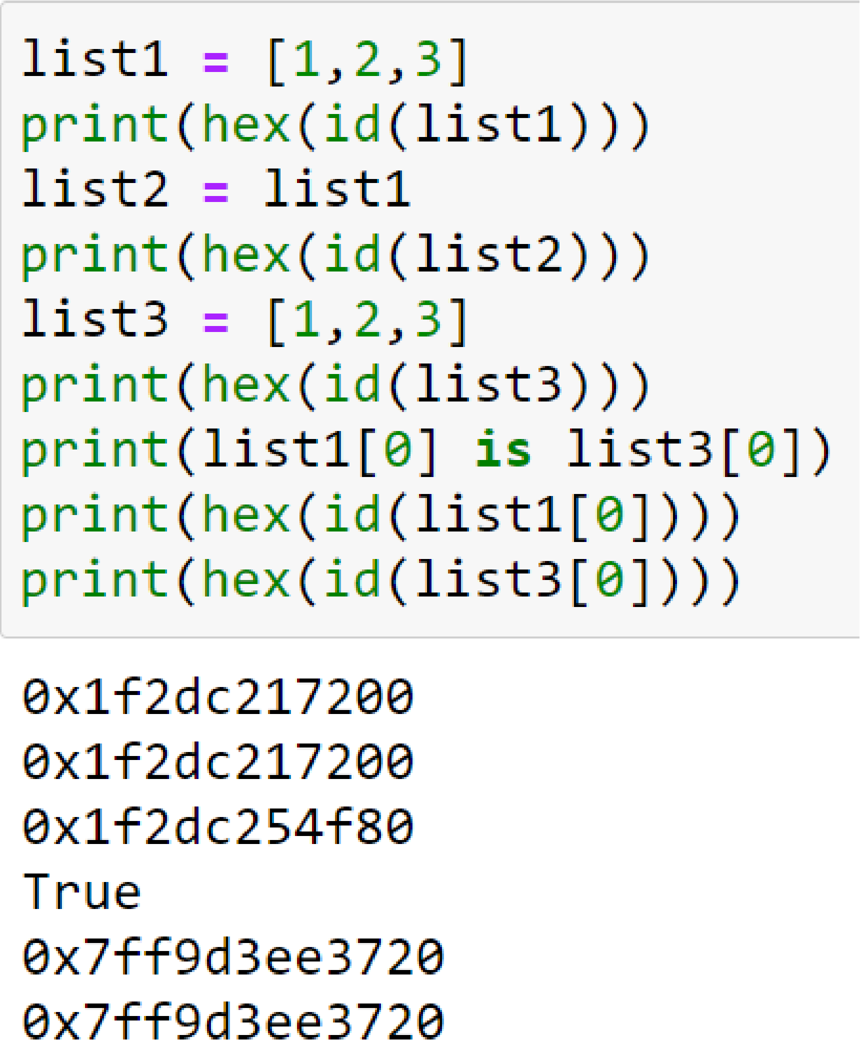


```
list1 = [1,2,3]
print(hex(id(list1)))
list1[0] = 97
print(list1)
print(hex(id(list1)))
# repeat assignment
list1 = [1,2,3]
print(hex(id(list1)))
```



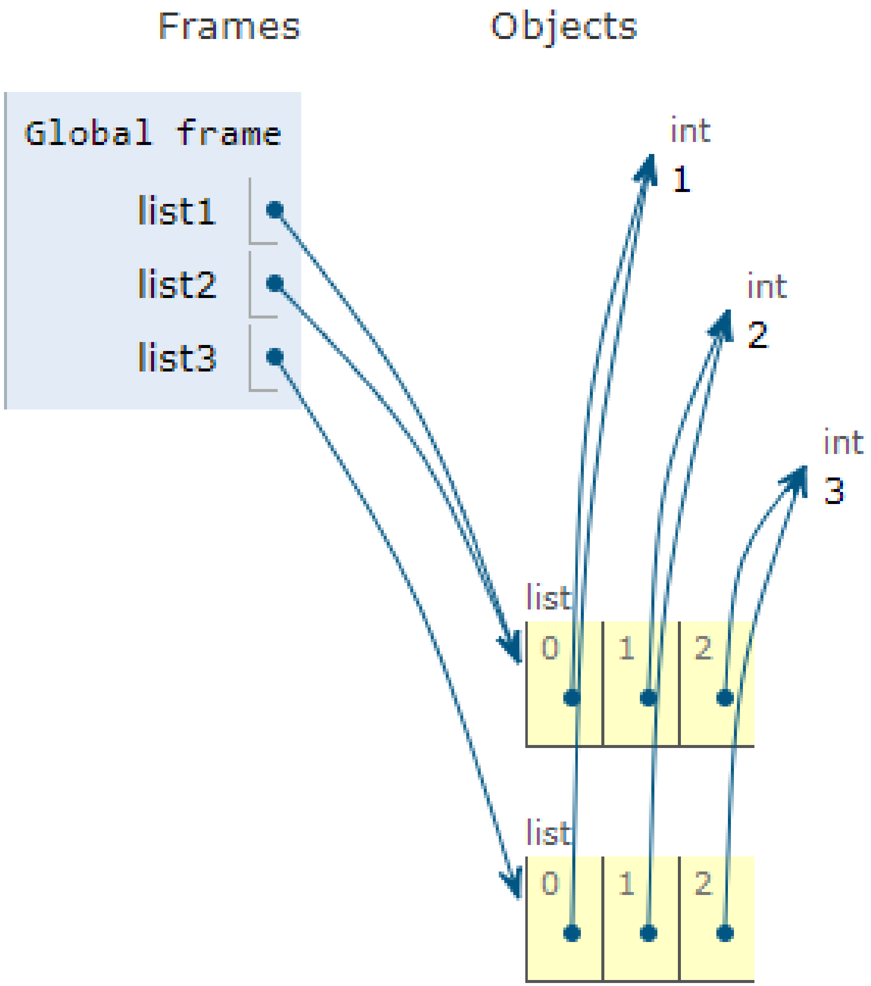
```
0x82015ba248
[97, 2, 3]
0x82015ba248
0x8202ed8208
```

One more look at mutable object



```python
list1 = [1,2,3]
print(hex(id(list1)))
list2 = list1
print(hex(id(list2)))
list3 = [1,2,3]
print(hex(id(list3)))
print(list1[0] is list3[0])
print(hex(id(list1[0])))
print(hex(id(list3[0])))
```

```
0x1f2dc217200
0x1f2dc217200
0x1f2dc254f80
True
0x7ff9d3ee3720
0x7ff9d3ee3720
```



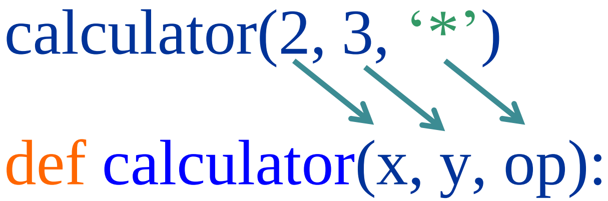
Memory model explained

Python’s memory model

- Equality and identity
- Assignments: mutable vs. immutable types
- **Passing arguments to functions**
- **The function stack**
- **Local versus global variables**

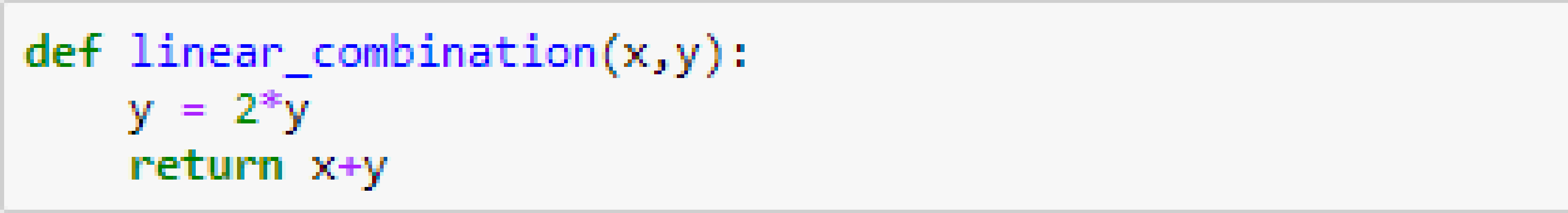
Passing arguments to functions

In a function’s call, **before** execution: arguments’ **values** are assigned to functions’ parameters **by order**

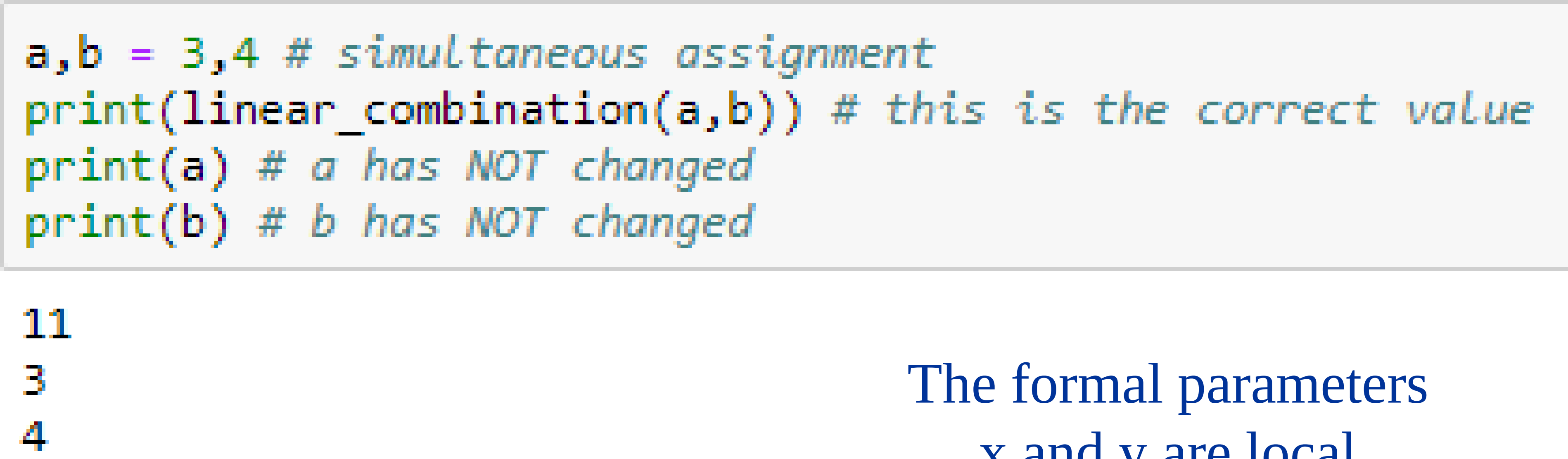


```
calculator(2, 3, ‘*’)

def calculator(x, y, op):
    if op == '+':
        return x+y
    elif …
    else:
        return None
```

Passing arguments to functions



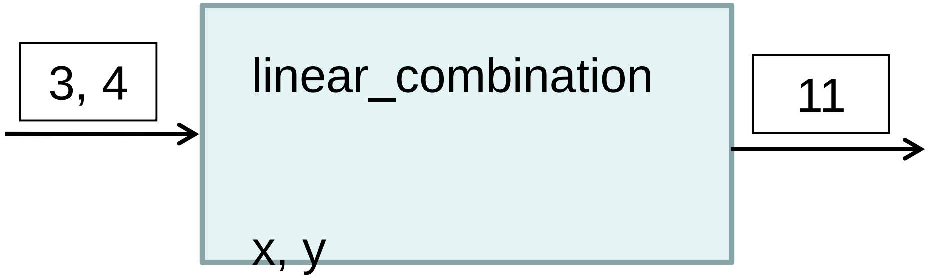
```python
def linear_combination(x,y):
    y = 2*y
    return x+y
```



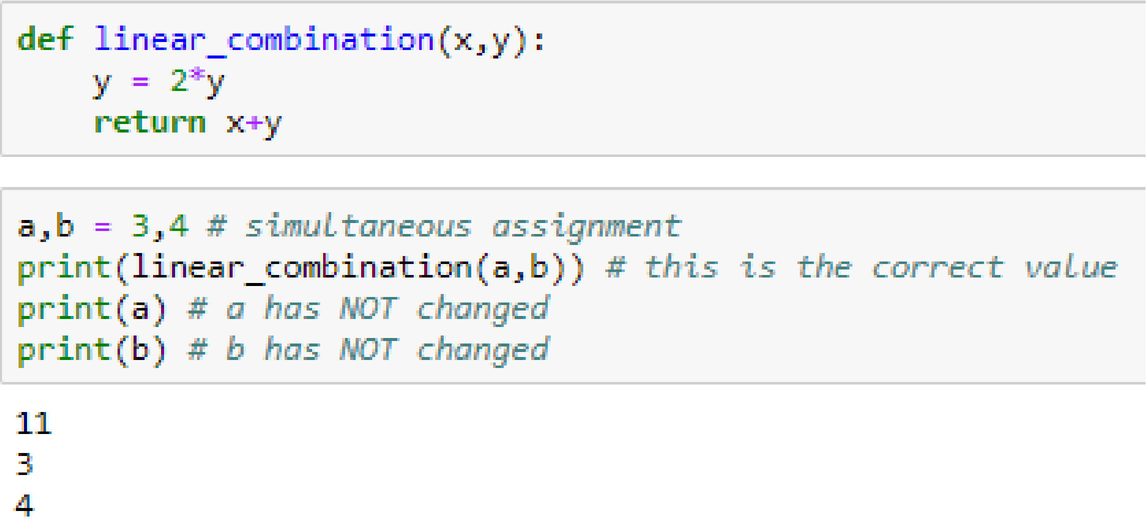
```python
a,b = 3,4 # simultaneous assignment
print(linear_combination(a,b)) # this is the correct value
print(a) # a has NOT changed
print(b) # b has NOT changed
```

```
11
3
4
```

The formal parameters
x and y are local



Passing arguments to functions



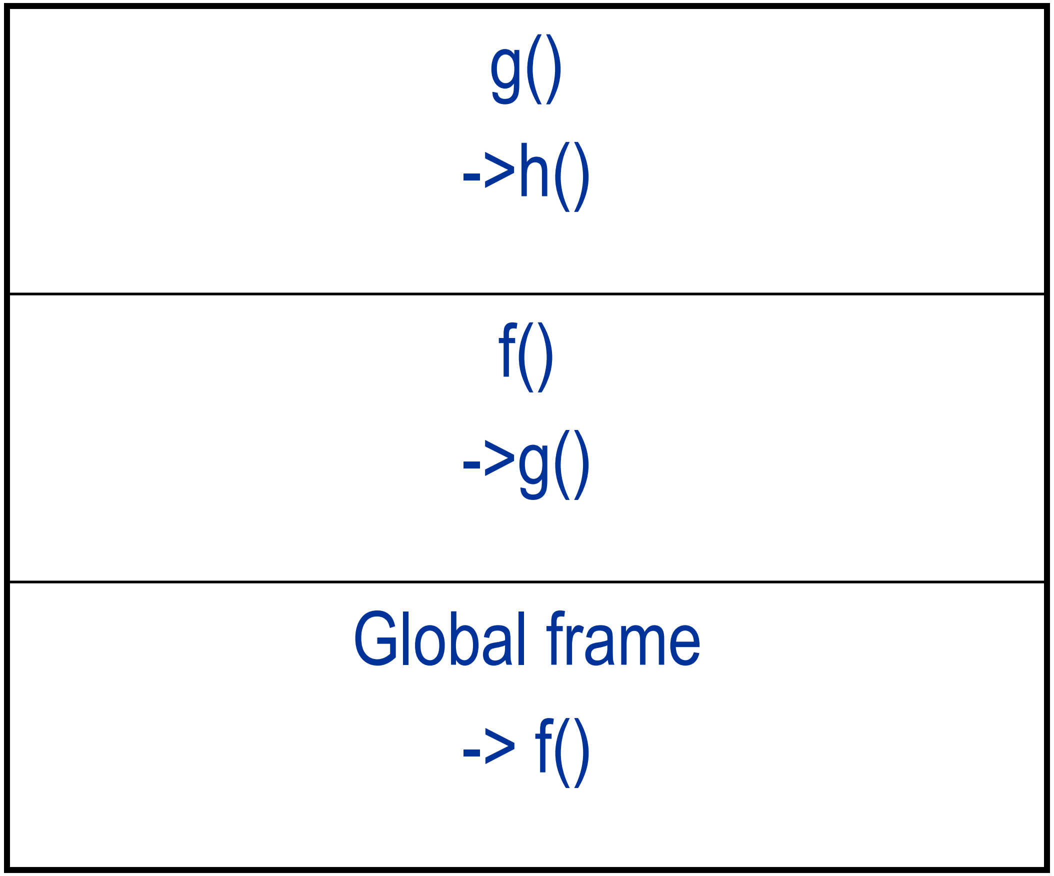
```python
def linear_combination(x,y):
    y = 2*y
    return x+y
```

```python
a,b = 3,4 # simultaneous assignment
print(linear_combination(a,b)) # this is the correct value
print(a) # a has NOT changed
print(b) # b has NOT changed
```

```
11
3
4
```

- Variables a,b are copied from the calling environment (global frame) to the function frame to variables named x,y (different address!)
- The assignment y=2*y changed y inside the body of linear combination(x,y). This change is kept local, inside the body of the function. The change is not visible by the calling environment.
- Visualize memory view with Pythontutor: https://goo.gl/V2yo4Z
- What if the function argument names are changed to a,b?

The function stack



g()

->h()

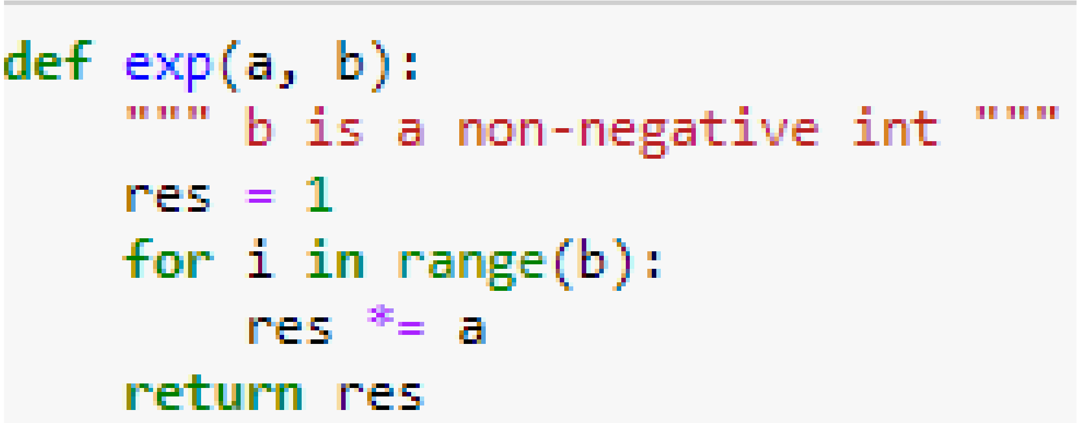
f()

->g()

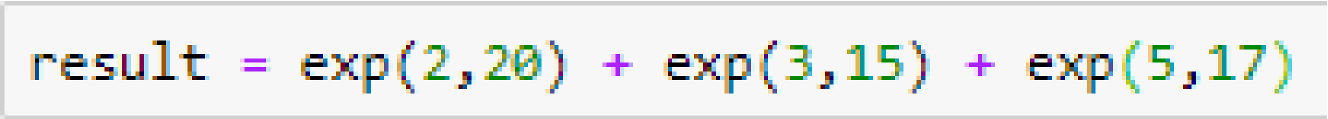
Global frame

-> f()

The call stack



```python
def exp(a, b):
    """ b is a non-negative int """
    res = 1
    for i in range(b):
        res *= a
    return res
```

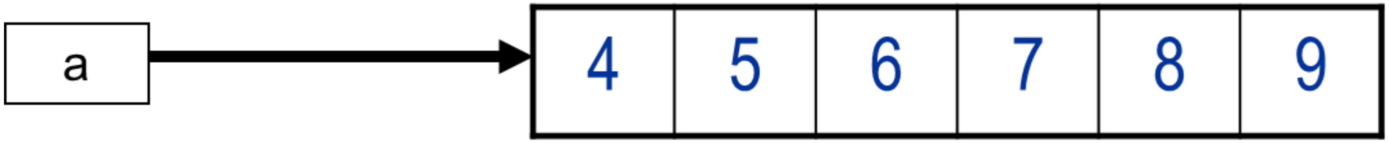


```python
result = exp(2,20) + exp(3,15) + exp(5,17)
```

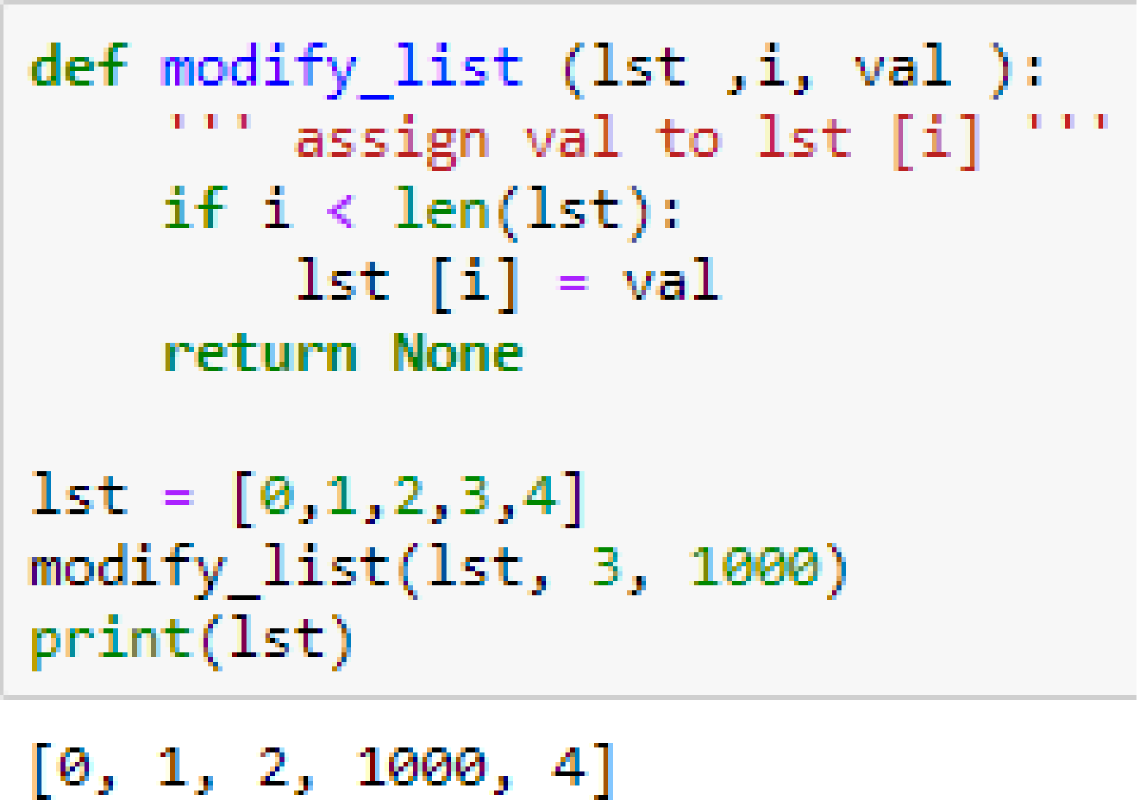
Visualize the call stack: https://goo.gl/ukWTdS

Passing arguments to functions

- The address of the actual parameters is passed to the corresponding formal parameters in the function.
- An assignment to the formal parameter within the function body creates a new object, and causes the formal parameter to address it.
- This change is not visible to the original caller's environment.
- Contents of mutable arguments (lists) can be changed within a function



Mutable objects as formal variables



```
def modify_list (lst ,i, val ):
    ''' assign val to lst [i] '''
    if i < len(lst):
        lst [i] = val
    return None

lst = [0,1,2,3,4]
modify_list(lst, 3, 1000)
print(lst)
```

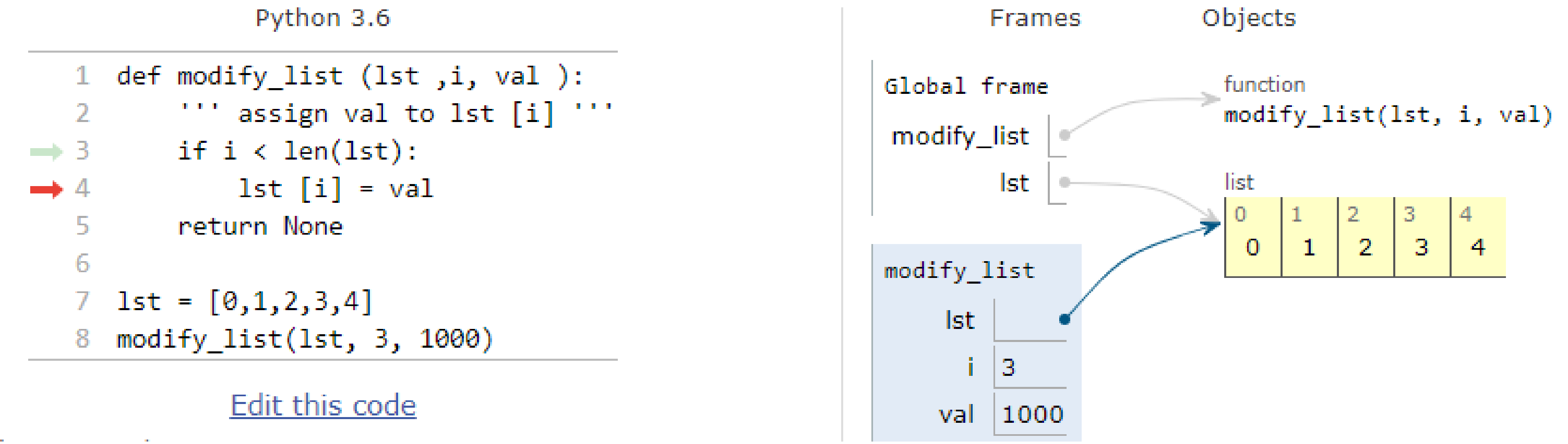
```
[0, 1, 2, 1000, 4]
```

- Mutating one of its parameters, the address in the function remains the same as in the calling environment

- Changes to the calling environment, that not caused through returned functions' values, are called side effects

Visualize

https://goo.gl/n3JDe9



Python 3.6

```
def modify_list (lst ,i, val ):
    ''' assign val to lst [i] '''
    if i < len(lst):
        lst [i] = val
    return None

lst = [0,1,2,3,4]
modify_list(lst, 3, 1000)
```

Edit this code

Frames

Objects

Global frame

modify_list

lst

function

modify_list(lst, i, val)

list

0	1	2	3	4
0	1	2	3	4

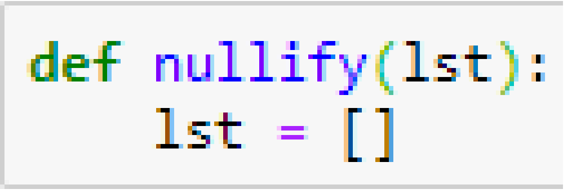
modify_list

lst

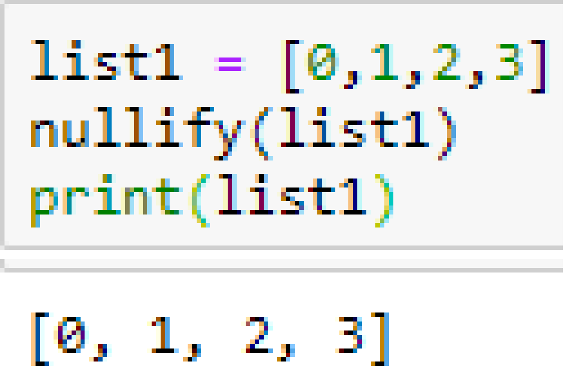
i 3

val 1000

Mutation vs. assignment in functions



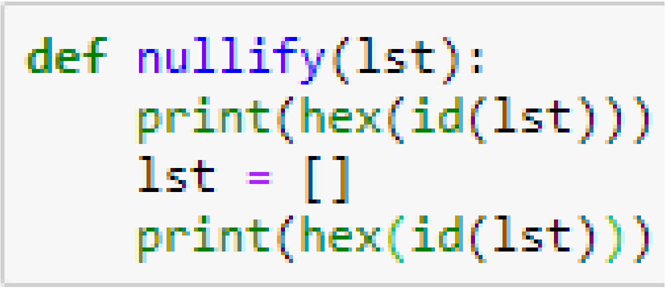
```
def nullify(lst):
    lst = []
```



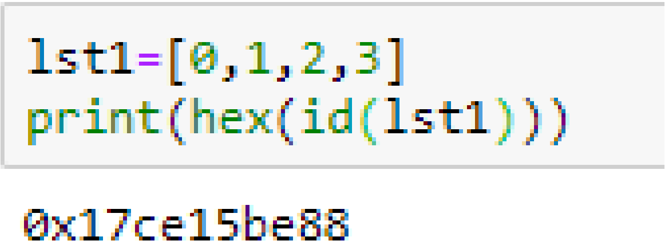
```
list1 = [0,1,2,3]
nullify(list1)
print(list1)
```

```
[0, 1, 2, 3]
```

2nd trial

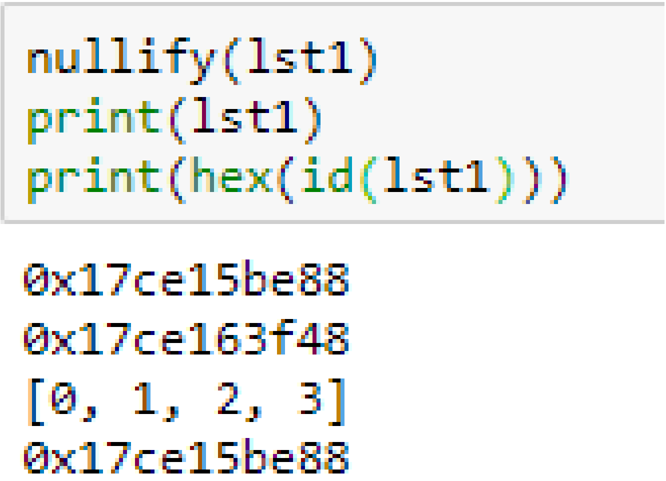


```
def nullify(lst):
    print(hex(id(lst)))
    lst = []
    print(hex(id(lst)))
```



```
lst1=[0,1,2,3]
print(hex(id(lst1)))
```

```
0x17ce15be88
```

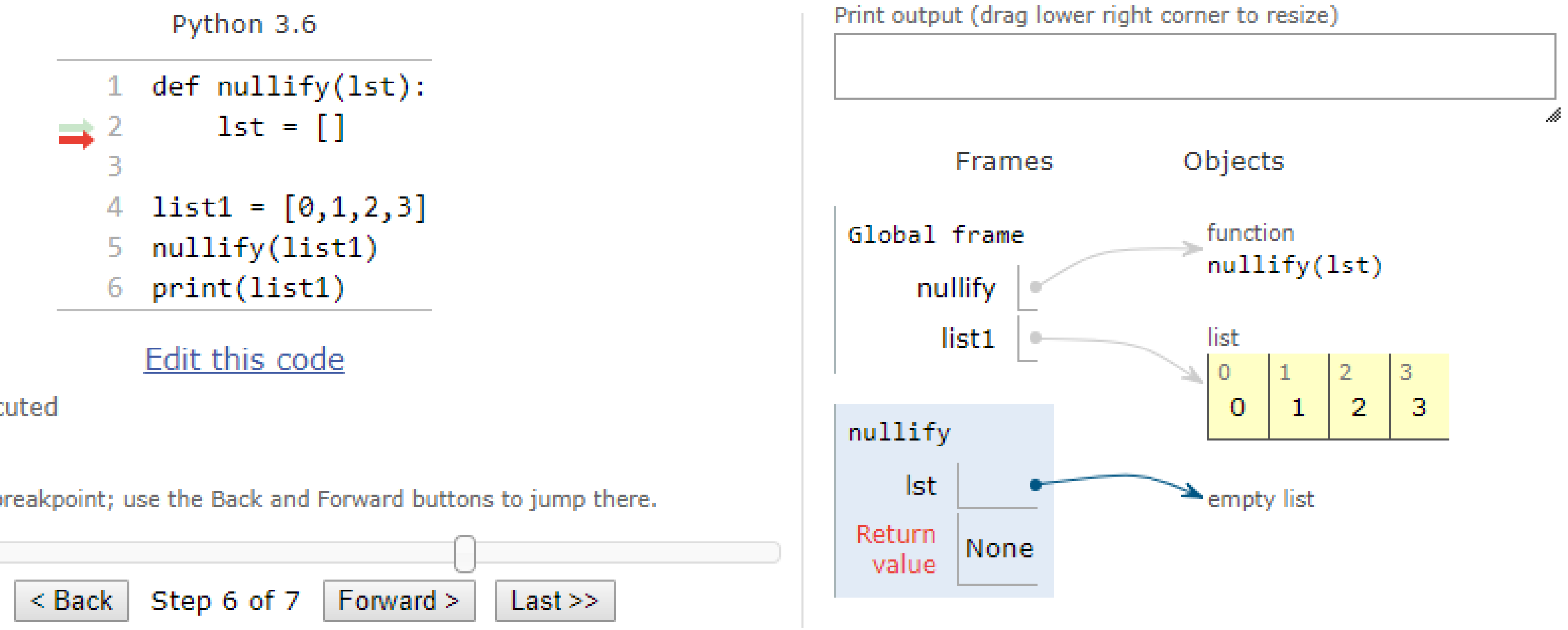


```
nullify(lst1)
print(lst1)
print(hex(id(lst1)))
```

```
0x17ce15be88
0x17ce163f48
[0, 1, 2, 3]
0x17ce15be88
```

Visualize

https://goo.gl/t6gTXi



Python 3.6

```
def nullify(lst):
    lst = []

list1 = [0,1,2,3]
nullify(list1)
print(list1)
```

Edit this code

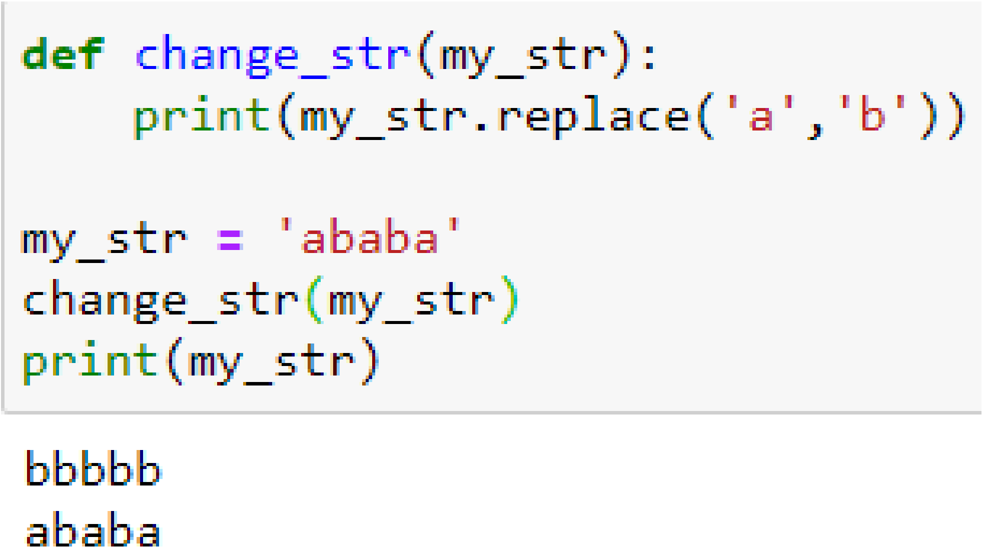
cuted

reakpoint; use the Back and Forward buttons to jump there.

Step 6 of 7

Print output (drag lower right corner to resize)

Another example, with strings

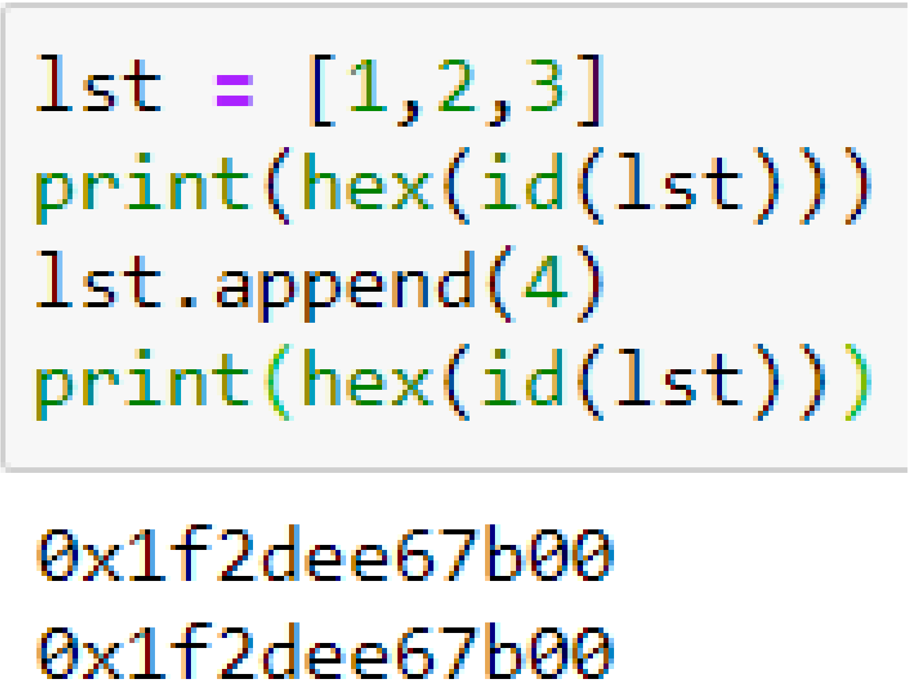


```python
def change_str(my_str):
    print(my_str.replace('a','b'))

my_str = 'ababa'
change_str(my_str)
print(my_str)
```

```
bbbbb
ababa
```

Python tutor

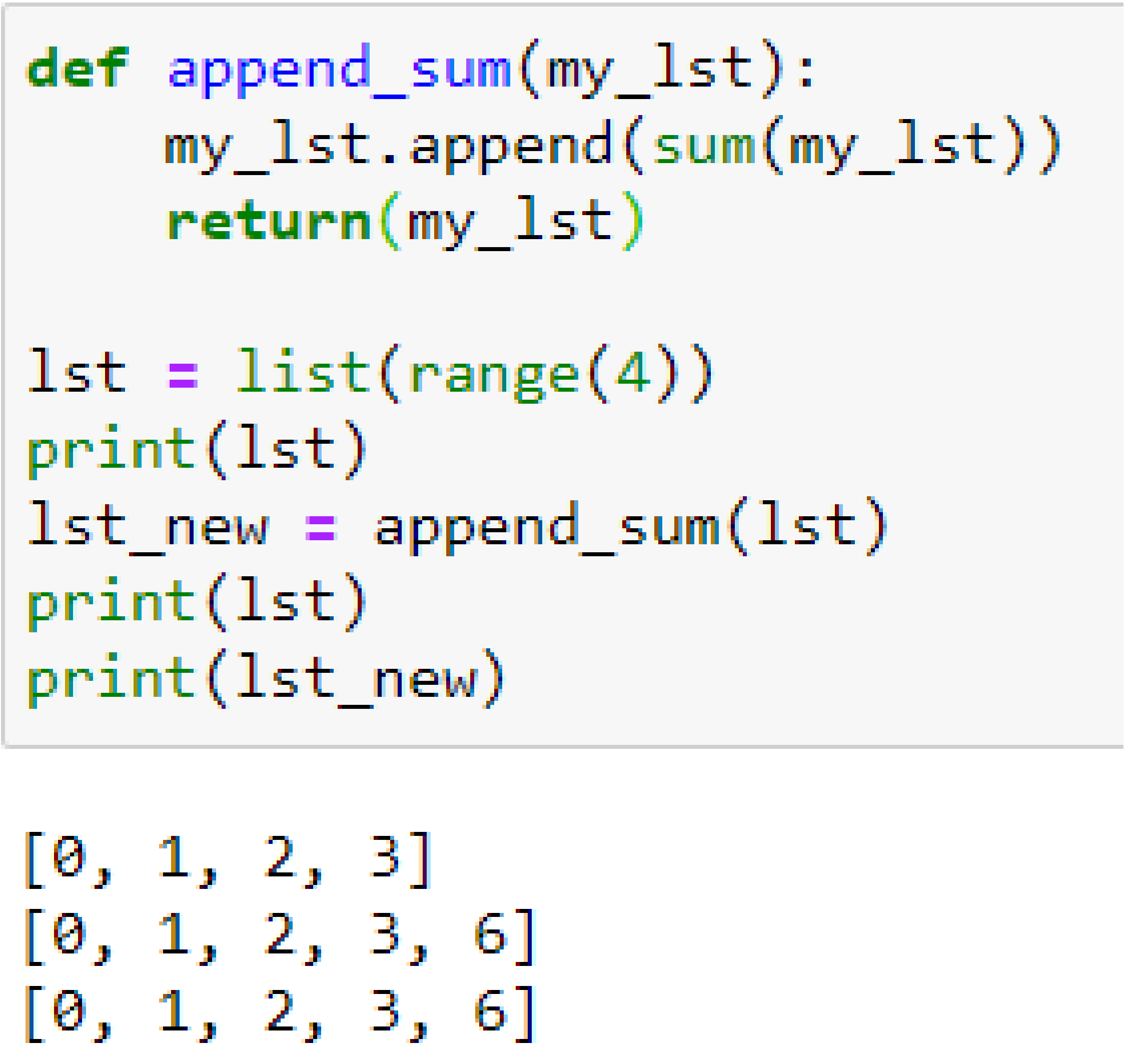
List append



```
lst = [1,2,3]
print(hex(id(lst)))
lst.append(4)
print(hex(id(lst)))
```

```
0x1f2dee67b00
0x1f2dee67b00
```

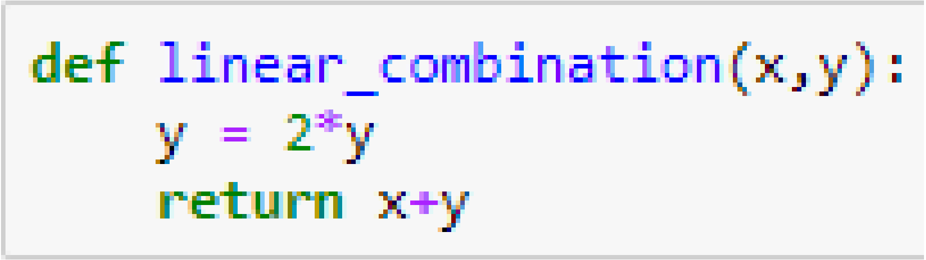
What is the output?



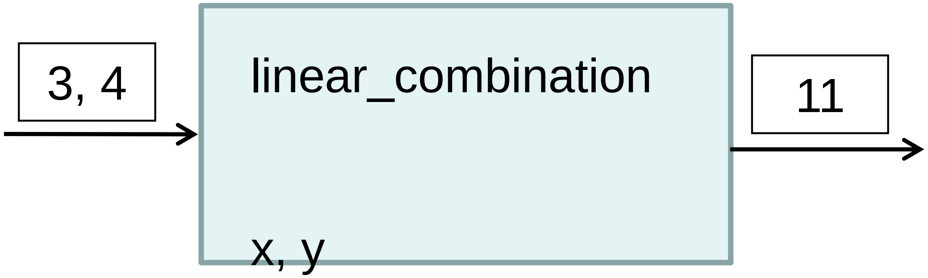
```python
def append_sum(my_lst):
    my_lst.append(sum(my_lst))
    return(my_lst)

lst = list(range(4))
print(lst)
lst_new = append_sum(lst)
print(lst)
print(lst_new)
```

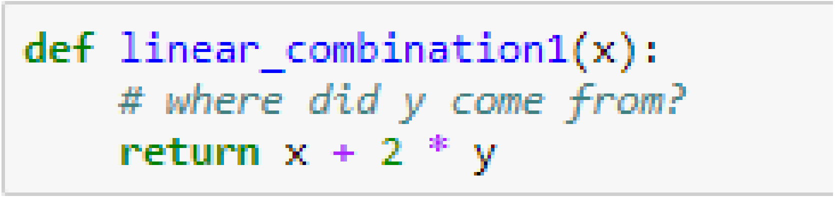
```
[0, 1, 2, 3]
[0, 1, 2, 3, 6]
[0, 1, 2, 3, 6]
```

Local vs. global variables

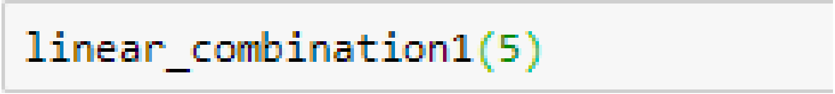


```
def linear_combination(x,y):
    y = 2*y
    return x+y
```

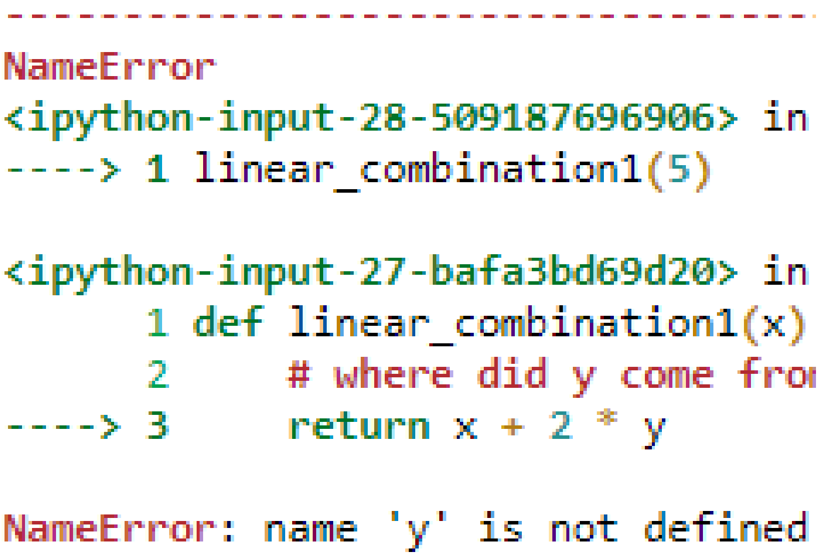




```
def linear_combination1(x):
    # where did y come from?
    return x + 2 * y
```



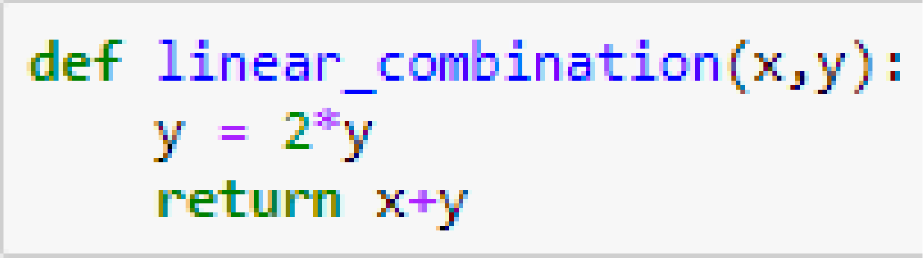
```
linear_combination1(5)
```



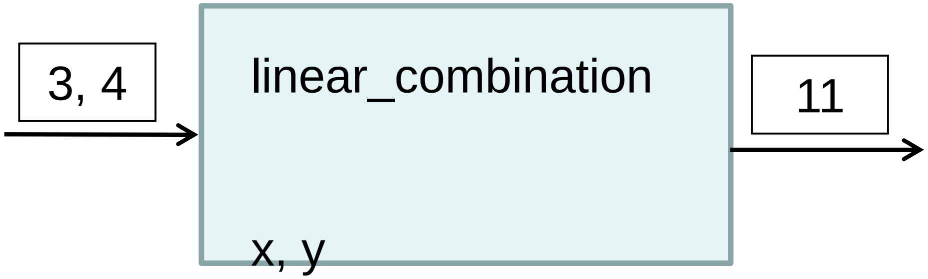
```
---------------------------------------
NameError
<ipython-input-28-509187696906> in
----> 1 linear_combination1(5)

<ipython-input-27-bafa3bd69d20> in
      1 def linear_combination1(x)
      2     # where did y come fro
----> 3     return x + 2 * y

NameError: name 'y' is not defined
```

Global variables are accessed inside a function but defined outside it

Local vs. global variables



```python
def linear_combination(x,y):
    y = 2*y
    return x+y
```

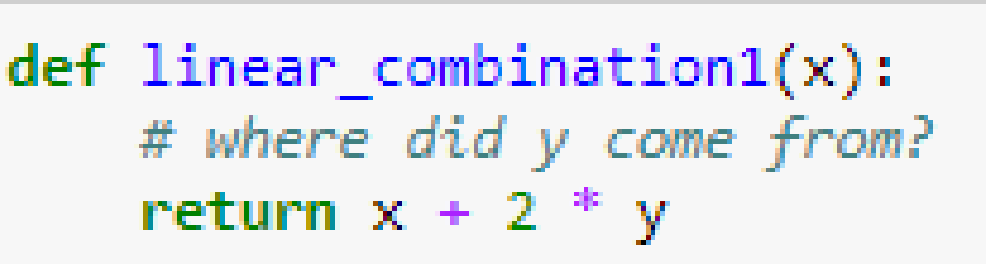


3, 4

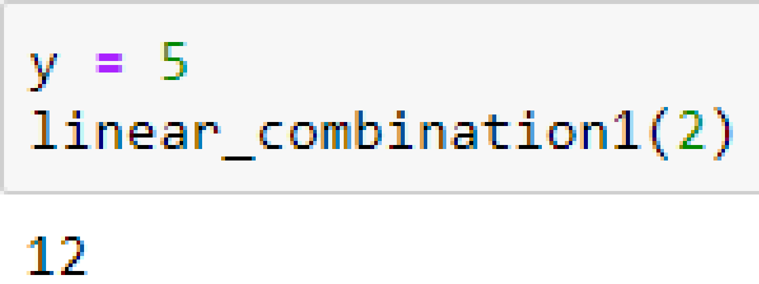
linear_combination

x, y

11

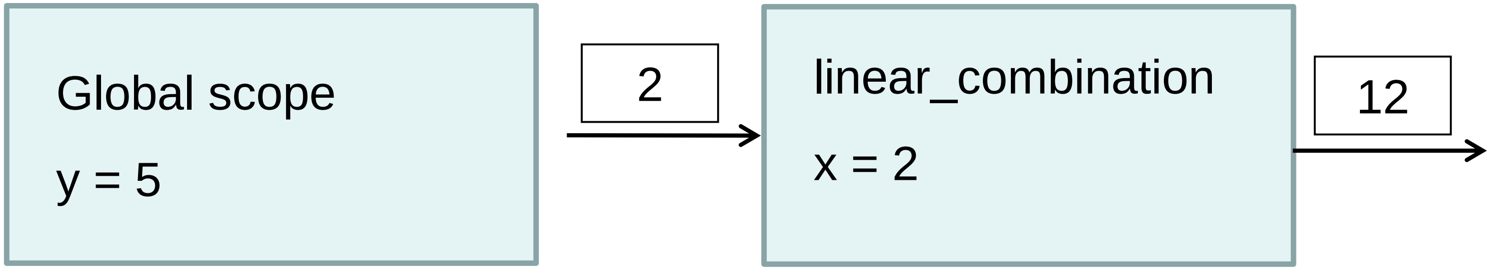


```python
def linear_combination1(x):
    # where did y come from?
    return x + 2 * y
```



```python
y = 5
linear_combination1(2)
```

```
12
```



Global scope

y = 5

2

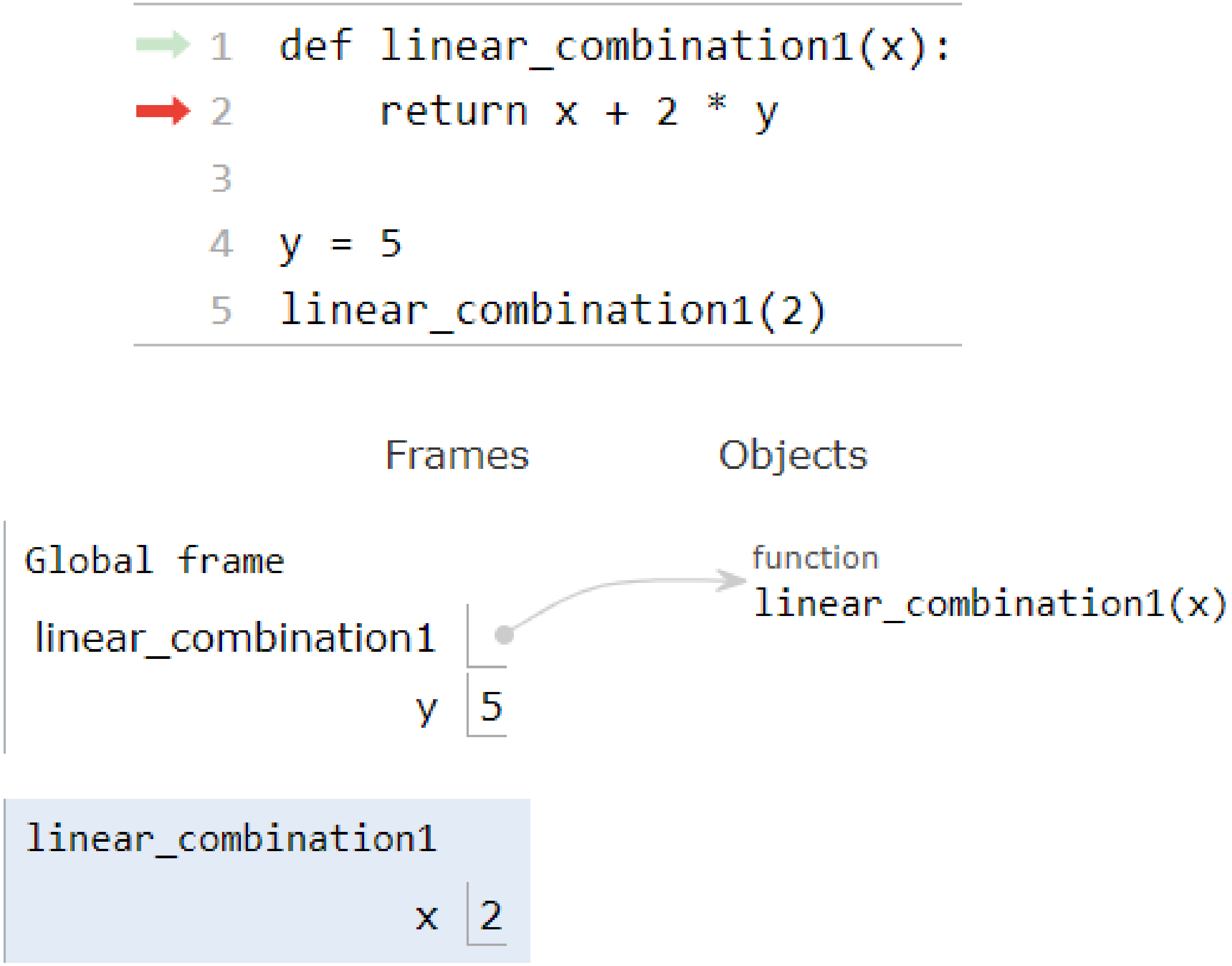
linear_combination

x = 2

12

Visualize

Python tutor



```
def linear_combination1(x):
    return x + 2 * y

y = 5
linear_combination1(2)
```

Frames

Objects

Global frame

linear_combination1

y 5

function
linear_combination1(x)

linear_combination1

x 2

Passing arguments to functions - summary

- Functions **cannot** change **immutable** objects sent to them (like numeric types or strings)
- Functions **can** change **mutable** objects sent to them (like lists). Changes made to these object will persist after the function ends
- Recommended reading: https://jeffknupp.com/blog/2012/11/13/is-python-callbyvalue-or-callbyreference-neither/
- Change the value of an outer variable:
  - By assignment of its return value
  - By accessing memory
  - By changing global variables (not advised)