15-112 Lecture 2

Lists

Instructor: Pat Virtue

Tuesday Logistics

As you walk in

Quiz will start at the beginning of lecture

- Have pencil/pen ready
- Silence phones

Quiz

Before we start

- Don’t open until we start
- Make sure your name and Andrew ID are on the front
- Read instruction page
- No questions (unless clarification on English)

Additional info

- 25 min

Announcements

Logistics changes related to Midterm 1 next week

- hw5 (due Sat 30-Sep at 8pm)
- Optional quiz5 (ungraded, due never)
- No pre-reading6
- Review for midterm (in-lecture next Tuesday)
- Thu 5-Oct: Midterm 1 (in-lecture next Thursday)

Stay tuned to Piazza for more details

Thursday Logistics

Announcements

Logistics changes related to Midterm 1 next week

- hw5 (due Sat 30-Sep at 8pm)
- Optional quiz5 (ungraded, due never)
- No pre-reading6
- Review for midterm (in-lecture next Tuesday)
- Thu 5-Oct: Midterm 1 (in-lecture next Thursday)

Stay tuned to Piazza for more details

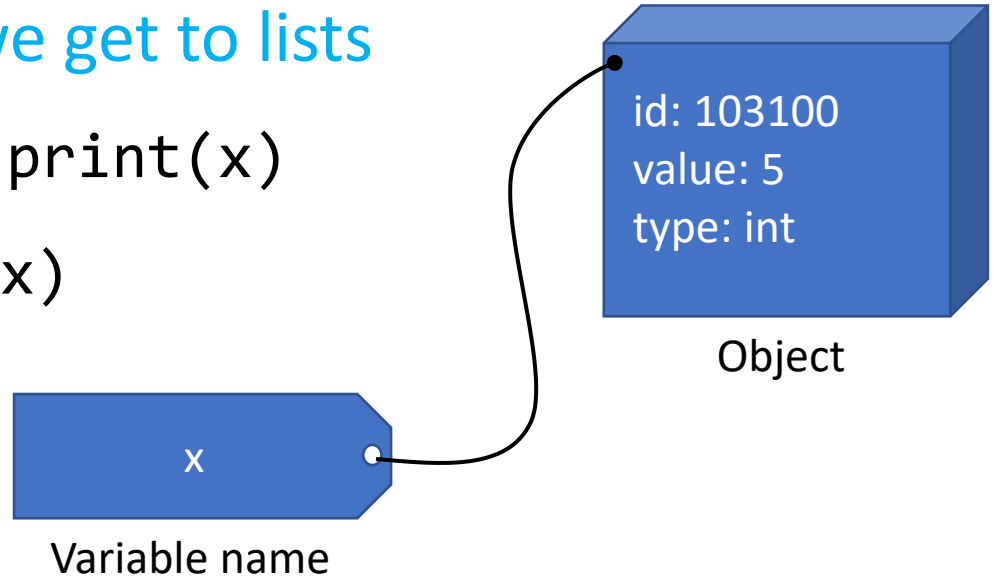
Lists

Python Objects and Variable Naming

All of the “things” in Python are objects

Python objects all have:

- id — More on object ids when we get to lists
- value — We can try to see this with `print(x)`
- type — We can see this with `type(x)`



Variable naming

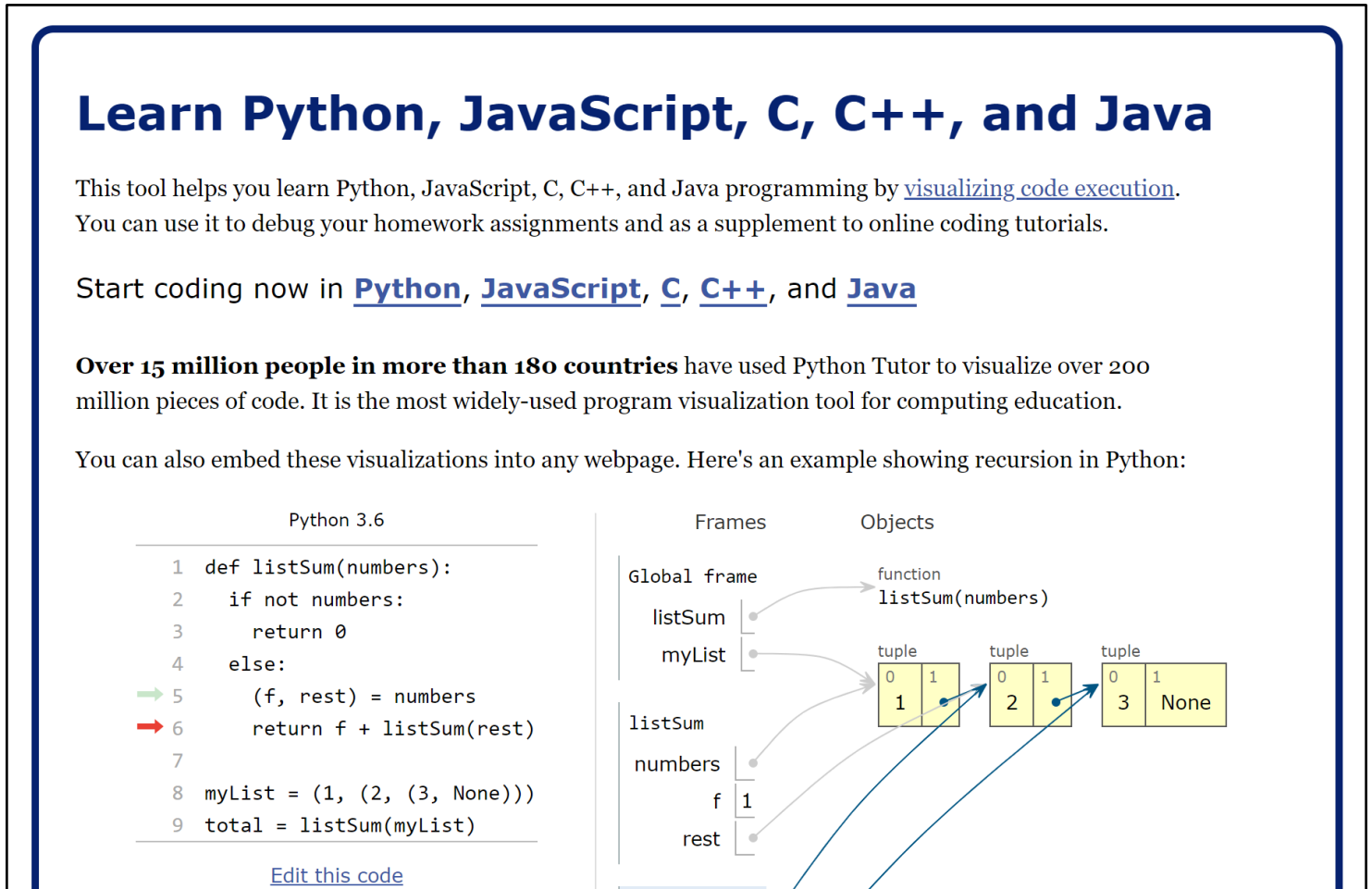
Think of a variable name as a gift tag attached to an object

Python keeps track of variable names to allow us to use that object later

Running Python

Pythontutor

- Help *see* how Python works

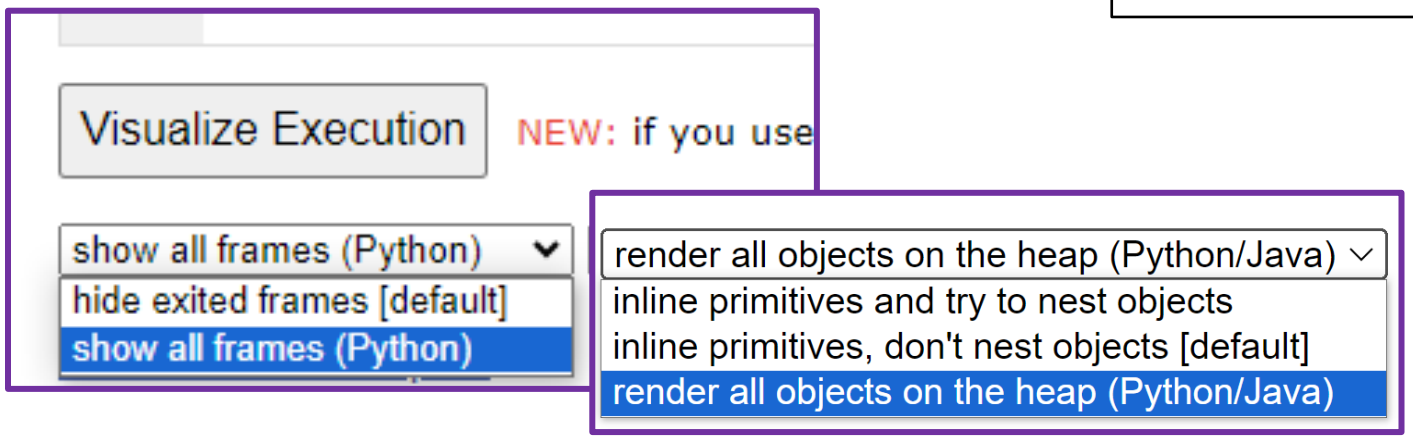


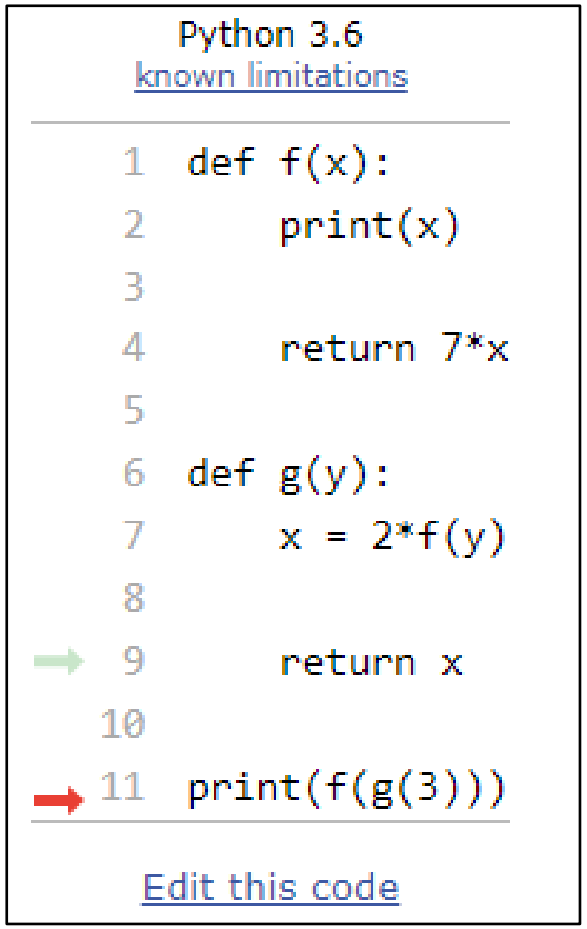
Running Python

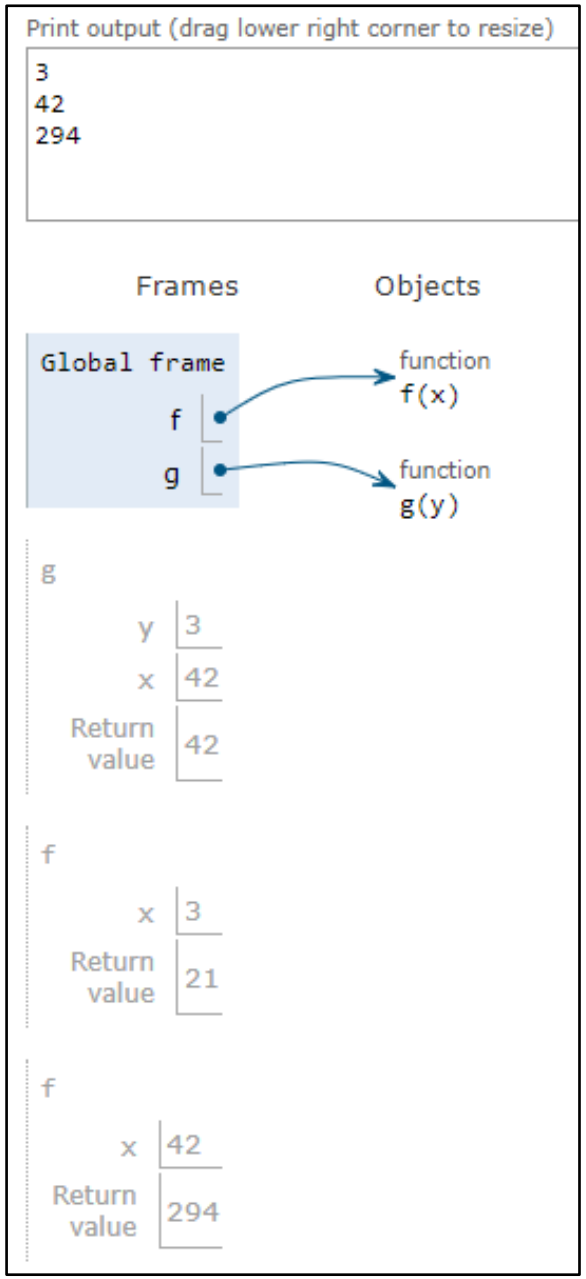
Pythontutor

- Help *see* how Python works
- Helpful to learn how to write out work for code tracing

Optional settings (bottom-left, bottom-center)



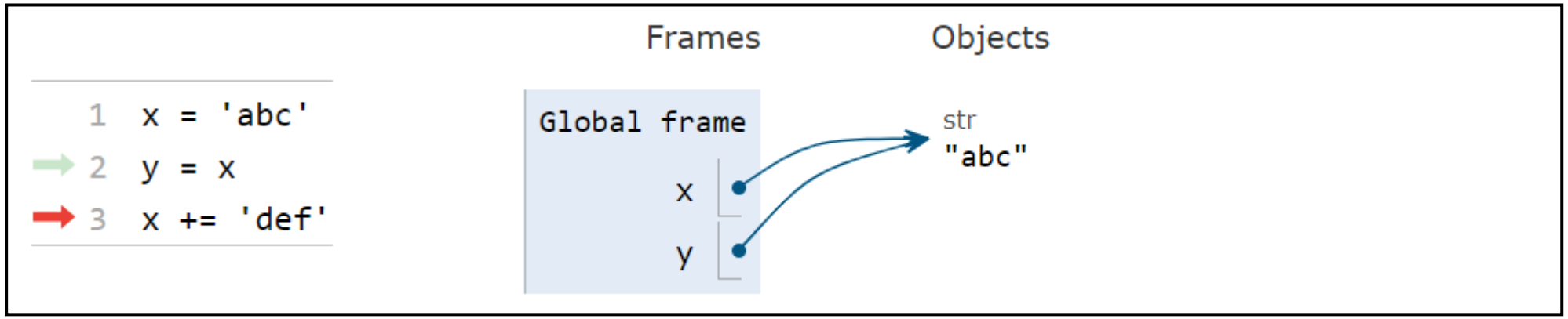


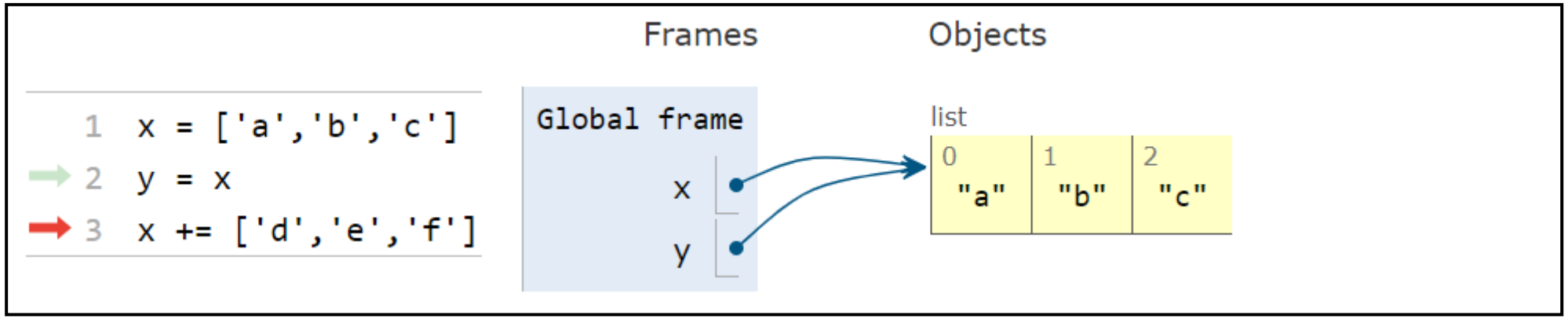


Strings vs Lists

Lists are mutable!

With strings, we always have to create a new string to modify an existing string



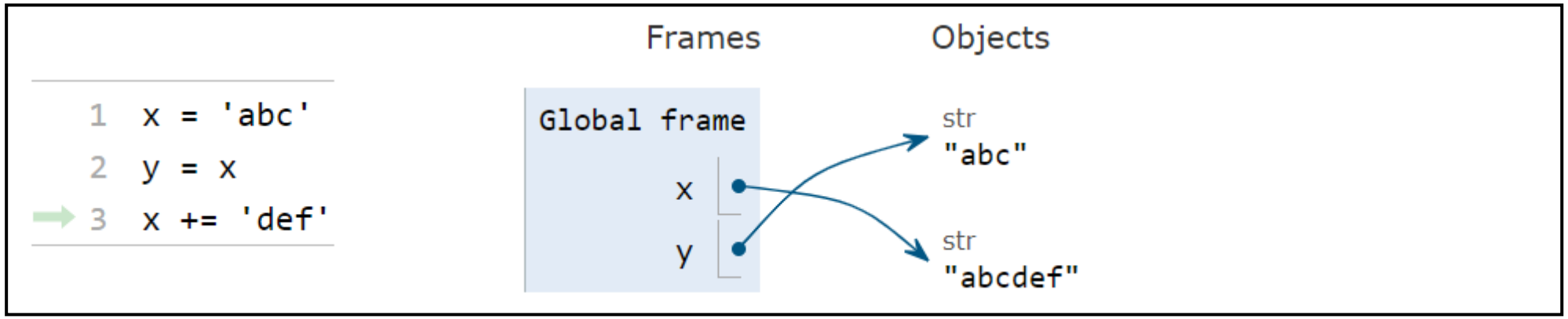


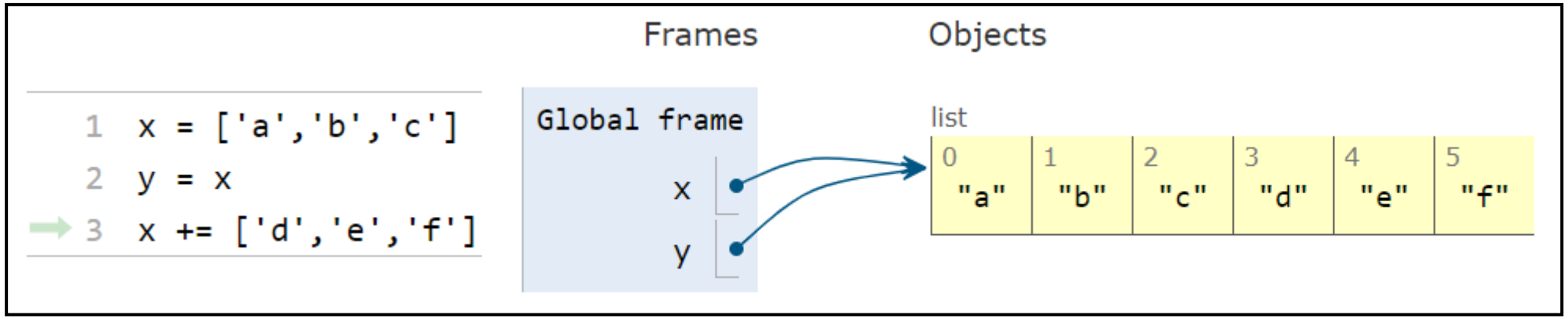
With lists, we can modify an existing list object

Strings vs Lists

Lists are mutable!

With strings, we always have to create a new string to modify an existing string





With lists, we can modify an existing list object

Reminder: Strings and aliases

Two variables are “aliases” are when they reference the exact same object

This happens when you assign a variable to another variable:

```
s = 'abc'
t = s
```

s and t are aliases referencing the same to the same exact string object 'abc’

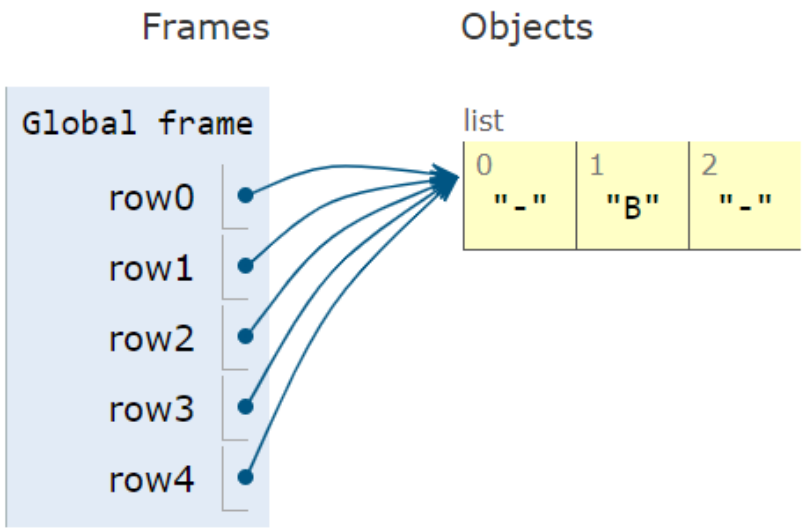
But…strings are immutable. We can’t possibly change s without making a new string.

```
s += 'def' # Assigns s to a new string 'abcdef’
# The string t is referencing remains 'abc'
```

Aliasing

Two variables are “aliases” are when they reference the exact same object

```
row0 = ['-','-','-']
row1 = row0
row2 = row0
row3 = row0
row4 = row0

row4[1] = 'B'

print(row0)
print(row1)
print(row2)
print(row3)
print(row4)
```



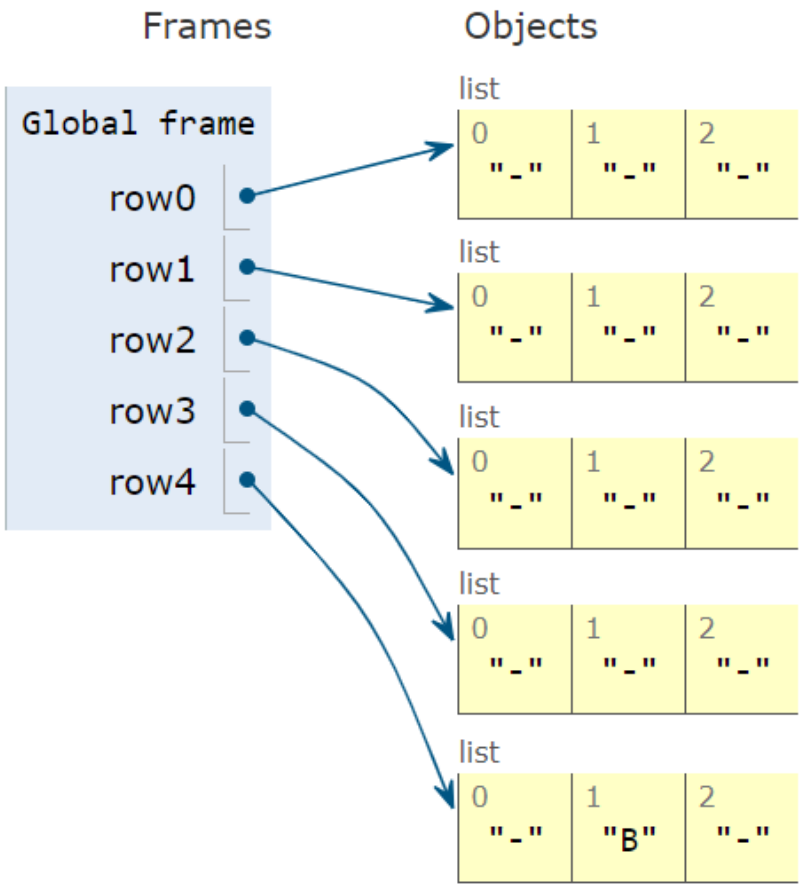
Print output (drag lower right corner to resize)

```
['-', 'B', '-']
['-', 'B', '-']
['-', 'B', '-']
['-', 'B', '-']
['-', 'B', '-']
```

Aliasing

Two variables are “aliases” are when they reference the exact same object

```
row0 = ['-','-','-']
row1 = ['-','-','-']
row2 = ['-','-','-']
row3 = ['-','-','-']
row4 = ['-','-','-']

row4[1] = 'B'

print(row0)
print(row1)
print(row2)
print(row3)
print(row4)
```



Print output (drag lower right corner to resize)

```
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
['-', 'B', '-']
```

Poll 1

What does this print?

```
import copy

A = [10, 20, 30]
B = A
C = copy.copy(A)

A[0] = 44
B[1] = 55
C[2] = 66

print('A:', A)
print('B:', B)
print('C:', C)
```

I. A: [44, 20, 30]
   B: [10, 55, 30]
   C: [10, 20, 66]

II. A: [44, 55, 30]
   B: [44, 55, 30]
   C: [10, 20, 66]

III. A: [44, 20, 66]
   B: [10, 55, 30]
   C: [44, 20, 66]

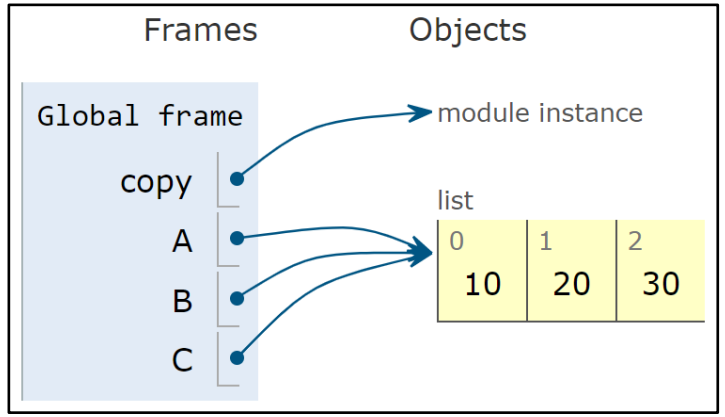
IV. A: [44, 55, 66]
   B: [44, 55, 66]
   C: [44, 55, 66]

Poll 2

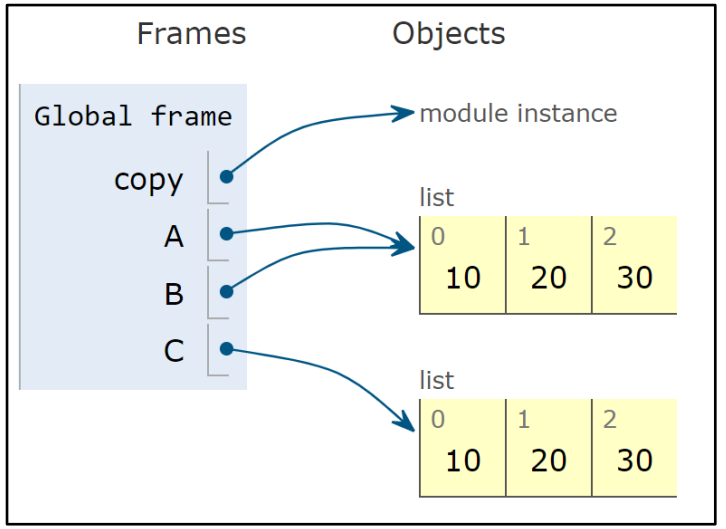
Which is the correct visualization?

```
import copy

A = [10, 20, 30]
B = A
C = copy.copy(A)
```

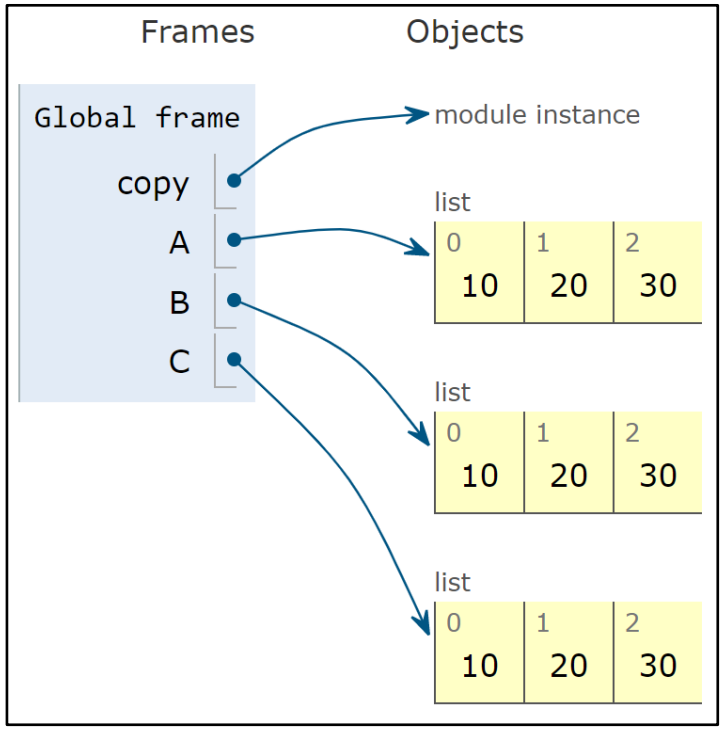
I.



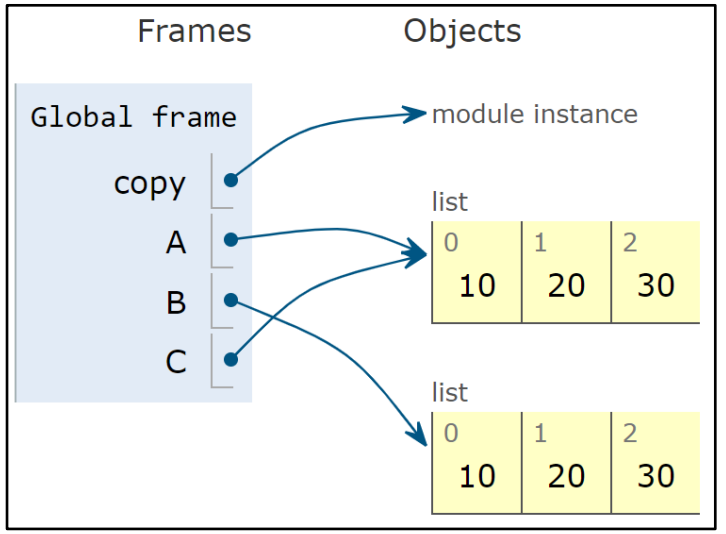
II.



III.



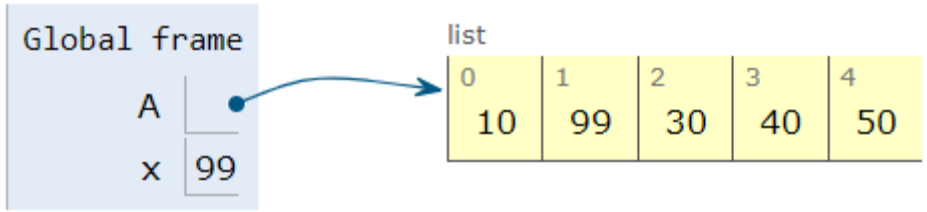
IV.



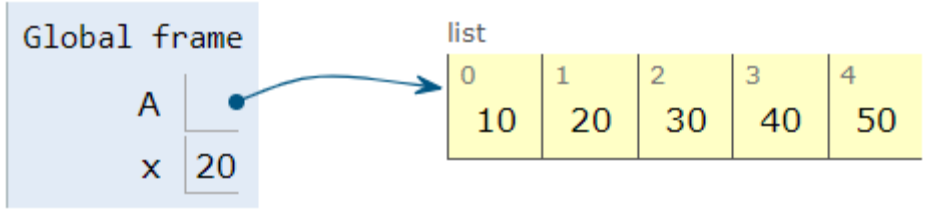
Reference slide

List indexing and slicing

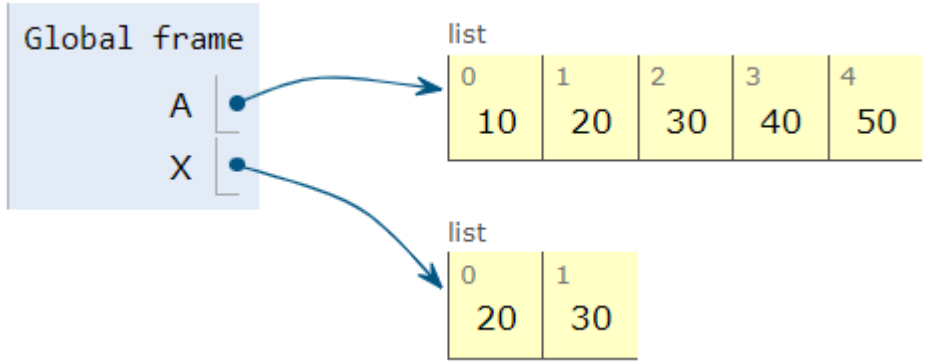
```
A = [10, 20, 30, 40, 50]
x = 99
A[1] = x
```



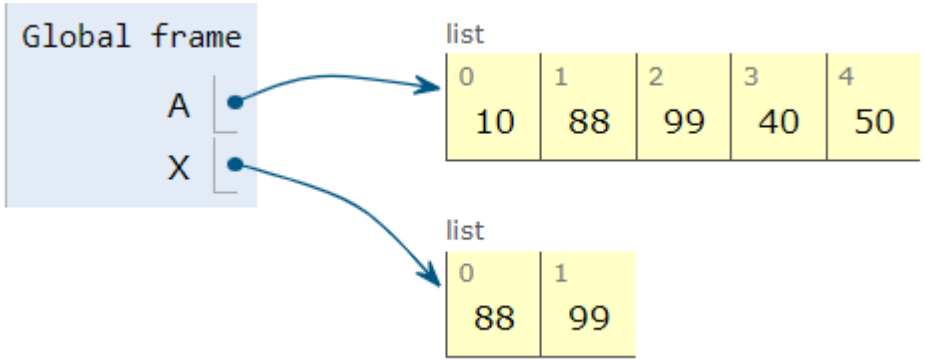

```
A = [10, 20, 30, 40, 50]
x = A[1]
```




```
A = [10, 20, 30, 40, 50]
X = A[1:3]
```



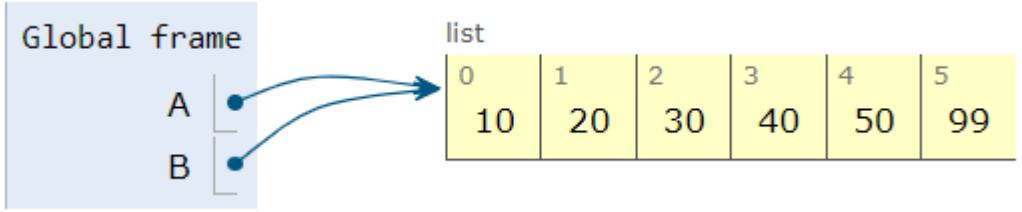

```
A = [10, 20, 30, 40, 50]
X = [88, 99]
A[1:3] = X
```



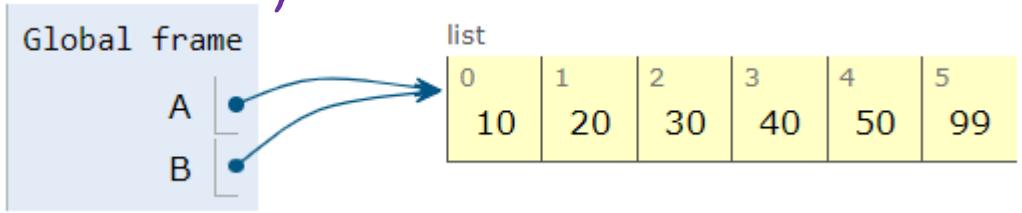
Adding elements

Reference slide

```
A = [10, 20, 30, 40, 50]
B = A
A.append(99)
```



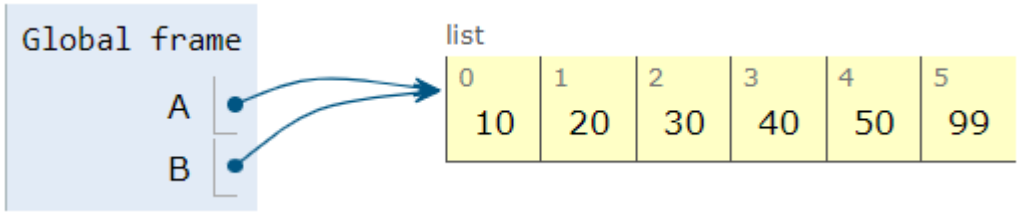

```
A = [10, 20, 30, 40, 50]
B = A
A += [99]
```



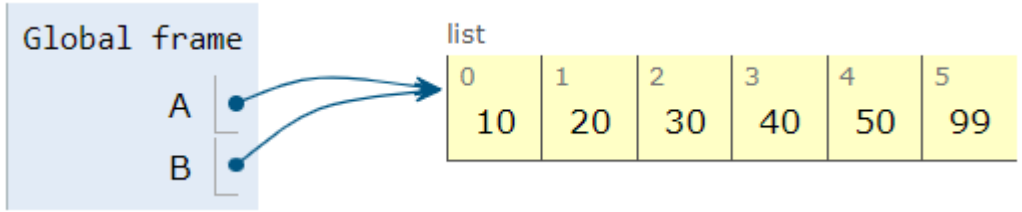
Adding elements

Reference slide

```
A = [10, 20, 30, 40, 50]
B = A
A.append(99)
```

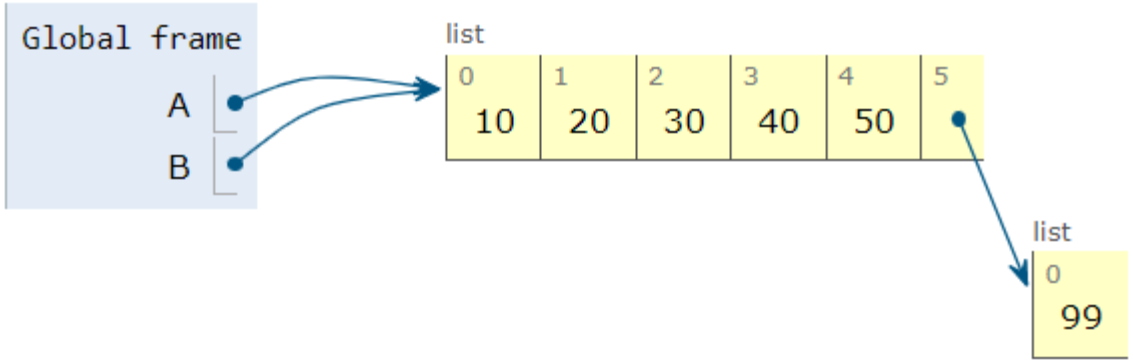



```
A = [10, 20, 30, 40, 50]
B = A
A += [99]
```

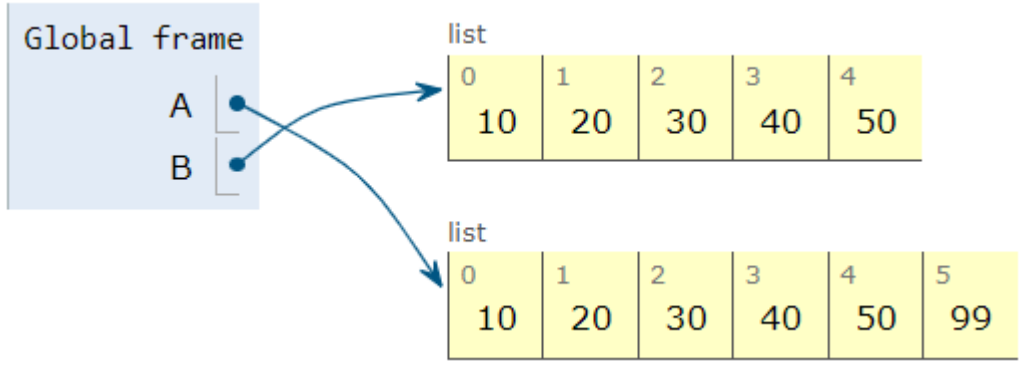



```
A = [10, 20, 30, 40, 50]
A += 99
TypeError: 'int' object is not iterable
```

```
# CAUTION
A = [10, 20, 30, 40, 50]
B = A
A.append([99])
```




```
# INCONSISTENT with +=
A = [10, 20, 30, 40, 50]
B = A
A = A + [99]
```



Poll 3

What are the resulting A, B, and C?

```
import copy

A = [10, 20, 30]
B = A
C = copy.copy(A)

A[0] = 44
B[1] = 55
C[2] = 66
A = A + [77]
```

I. A: [44, 20, 30, 77]
   B: [10, 55, 30]
   C: [10, 20, 66]

II. A: [44, 55, 30, 77]
   B: [44, 55, 30]
   C: [10, 20, 66]

III. A: [44, 20, 66, 77]
   B: [10, 55, 30]
   C: [44, 20, 66]

IV. A: [44, 55, 30, 77]
   B: [44, 55, 30, 77]
   C: [10, 20, 66]

Poll 4

What does this print?

```
def f(L):
    L.remove(3)

A = [2, 3, 4, 5]
print(f(A))
```

I. [2, 3, 4, 5]

II. [2, 4, 5]

III. [2, 3, 5]

IV. []

V. None

Poll 5

What does this print?

```
def f(L):
    L.remove(3)

A = [2, 3, 4, 5]
f(A)
print(A)
```

I. [2, 3, 4, 5]

II. [2, 4, 5]

III. [2, 3, 5]

IV. []

V. None

Reference slide

4.2.12 Summary of List Methods and Functions

Some mutating vs. non-mutating list analogs, for:

```
a = ['cat','dog','pig','cow']
```

Mutating (aliasing)	Non-mutating
`b = a`	`b = copy.copy(a)`
	`b = a[:]`
	`b = a + []`
	`b = list(a)`
`a.append('axolotl') # just the elem`	`a = a + ['axolotl']`
`a.extend(['axolotl']) # watch the brackets`	
`a += ['axolotl'] # also needs br`	

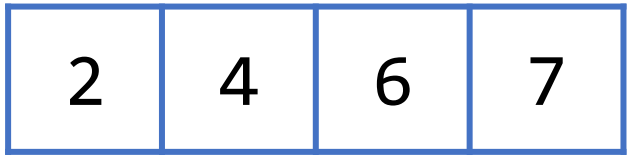
Caution: Mutating in Loops

Guided Exercise: removeEvens

Broken version:

```
for i in range(len(L)):
    if L[i] % 2 == 0:
        L.pop(i)
```

L = [2, 4, 6, 7]



Caution: Mutating in Loops

Guided Exercise: removeEvens

Broken version:

```
for i in range(len(L)):
    if L[i] % 2 == 0:
        L.pop(i)
```

Corrected version:

```
i = 0
while i < len(L):
    if L[i] % 2 == 0:
        L.pop(i)
    else:
        i += 1
```

```
L = [2, 4, 6, 7]
```

2	4	6	7

```
i = 0
L.pop(i)
```

4	6	7

```
i = 1
L.pop(i)
```

4	7

```
i = 2
L[i] % 2
```

Error: Index out of range

Poll 6

Which is best?

I.

```
def doubleValues(L):
    for i in range(len(L)):
        L[i] *= 2
```

II.

```
def doubleValues(L):
    for i in range(len(L)):
        L[i] *= 2
    return L
```

III.

```
def doubleValues(L):
    A = []
    for item in L:
        A.append(item*2)
    return A
```

IV.

```
def doubleValues(L):
    A = []
    for item in L:
        A.append(item*2)
```

Pattern: Building up a result

From Strings Lecture

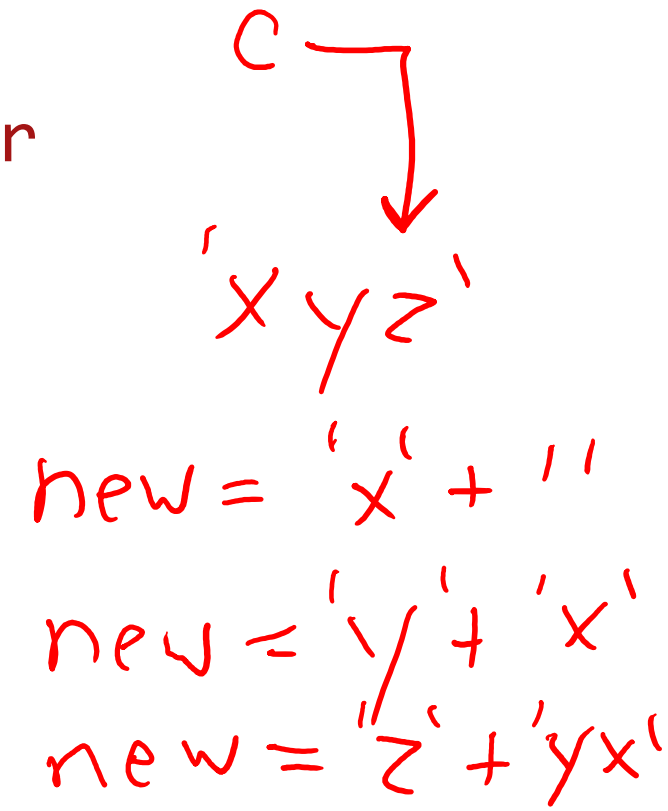
Building up a string

Sketch:

- Start with empty string: `result = ''`
- Loop
  - adding to string as needed: `result += nextChar`

Example:

```
def reverseString(s):
    newString = ''
    for c in s:
        newString = c + newString

    return newString
```



Pattern: Building up a result

Building up a string

Sketch:

- Start with empty string: `result = []`
- Loop
  - adding to string as needed: `result.append(nextVal)`

Example:

```
def doubleListValues(L):
    newList = []
    for val in L:
        newList.append(2*val)

    return newList
```

Poll 7 (unused)

What does this print?

```
def f():
    return 'a', 3

x = f()
print(type(x))
```

I. <class ‘int'>
II. <class ‘str’>
III. <class ‘list’>
IV. <class 'tuple’>
V. (<class ‘str’>, <class ‘int’>)
VI. ERROR
VII. I have no idea

Tuples and List Comprehensions

Reference slide

Tuples

Like lists but immutable

FAIL: `myTuple[0] = 99`

Simulate multiple return values

```
def sumProd(x, y):
    return x+y, x*y
```

Multiple assignment

```
cx, cy = width/2, height/2
```

One line swapping!

```
y, x = x, y
```

Single element tuples

```
myTuple = (99,)
```

List Comprehension

Sample for loop

```
newList = []
for variable in sequence:
    newList.append(expression)
```

Python shorthand

```
newList = [expression for variable in sequence]
```

Fashionable python

Image credit: https://www.nytimes.com/2020/09/21/style/exotic-skins-fashion-covid.html

Reference slide

List Comprehension

Sample for loop (now with a filter)

```
newList = []
for variable in sequence:
    if condition:
        newList.append(expression)
```

Python shorthand (now with a filter)

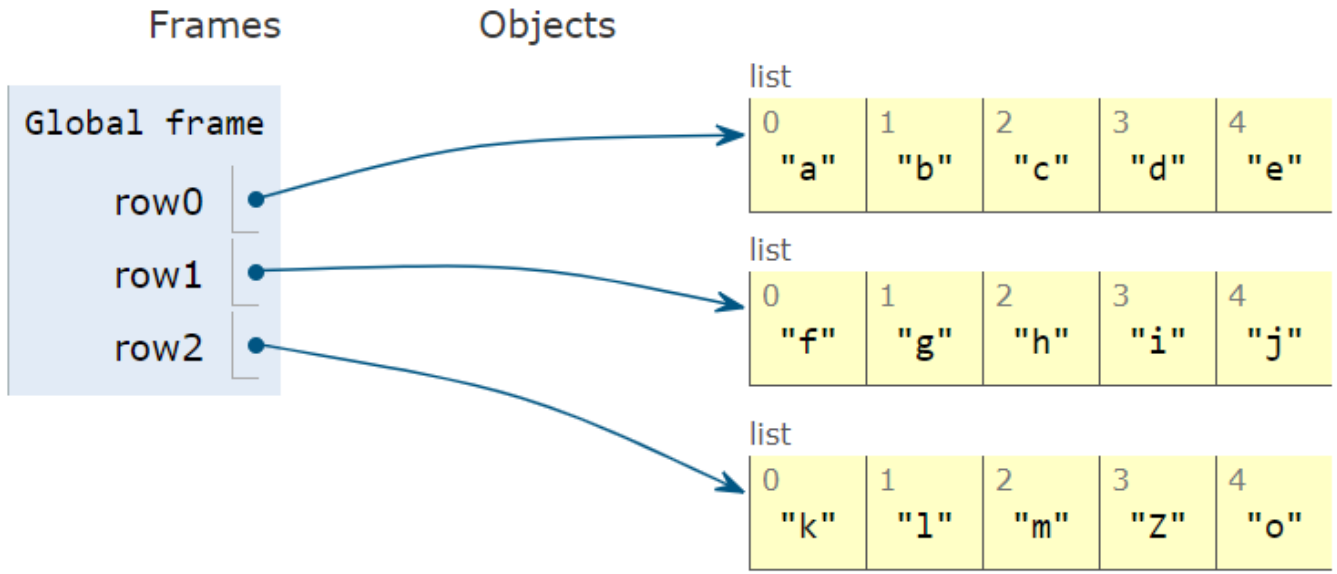
```
newList = [expression for variable in sequence if condition]
```

Image credit: https://www.nytimes.com/2020/09/21/style/exotic-skins-fashion-covid.html

2D Lists

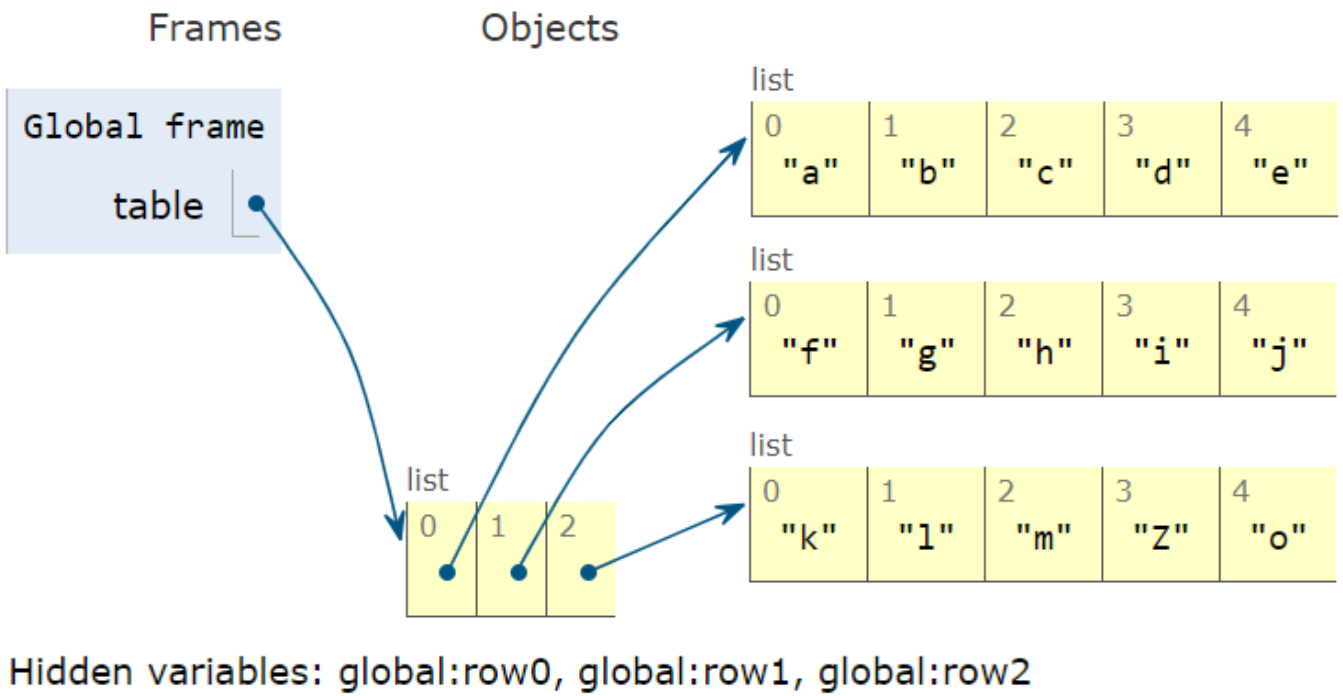
We can put lists inside elements of a list

```
row0 = ['a','b','c','d','e']
row1 = ['f','g','h','i','j']
row2 = ['k','l','m','n','o']

row2[3] = 'Z'
```



We can put lists inside elements of a list

```
row0 = ['a','b','c','d','e']
row1 = ['f','g','h','i','j']
row2 = ['k','l','m','n','o']

table = [row0, row1, row2]
table[2][3] = 'Z'
```



Hidden variables: global:row0, global:row1, global:row2

Traversing 2D Lists

Printing rectangular list

```
# Create rectangular 2D list
table = [[900, 901, 902],
         [910, 911, 912],
         [920, 921, 922]]

numRows = len(table)
numCols = len(table[0]) # Assume all rows have the same width

for i in range(numRows):
    for j in range(numCols):
        value = table[i][j]
        print(value, end=',') # Print on same row (with commas)
    print() # New line after row
```

Traversing 2D Lists

Printing non-rectangular (irregular) (ragged) list

```
# Create non-rectangular 2D list
table = [[900, 901],
         [910, 911, 912, 913, 914],
         [920, 921, 922]]

numRows = len(table)

for i in range(numRows):
    numCols = len(table[i])

    for j in range(numCols):
        value = table[i][j]
        print(value, end=',')
    print() # New line after row
```

```
# Simpler if we don't need indices
for row in table:
    for value in row:
        print(value, end=',')
    print() # New line after row
```

Creating 2D Lists

Creating 2D Lists

If you know the values, you can just type out the list of lists

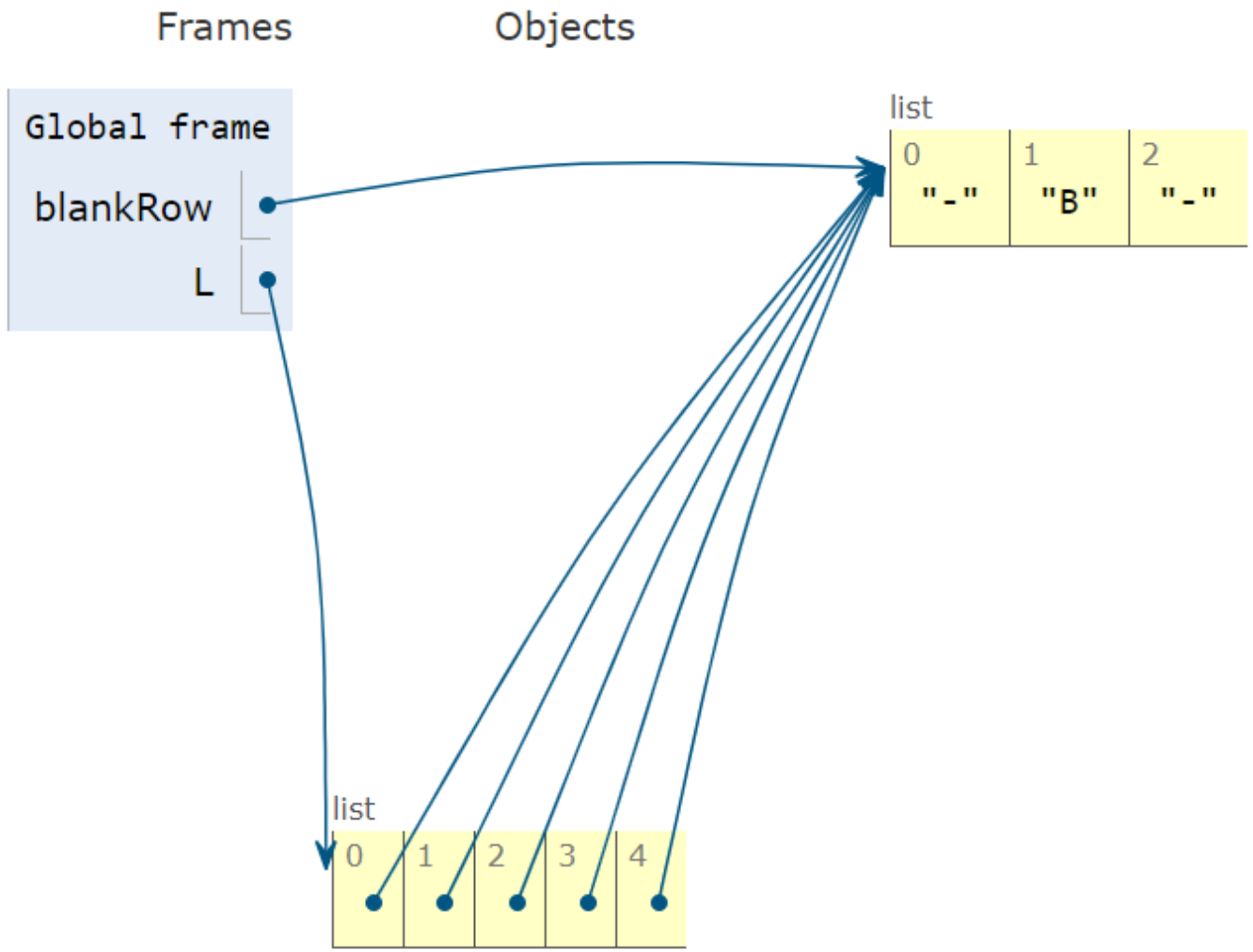
```
data = [[900, 901, 902], [910, 911, 912], [920, 921, 922]]
```

```
# Same as above but code is easier to read
data = [[900, 901, 902],
        [910, 911, 912],
        [920, 921, 922]]
```

Aliasing

Two variables are “aliases” are when they reference the exact same object

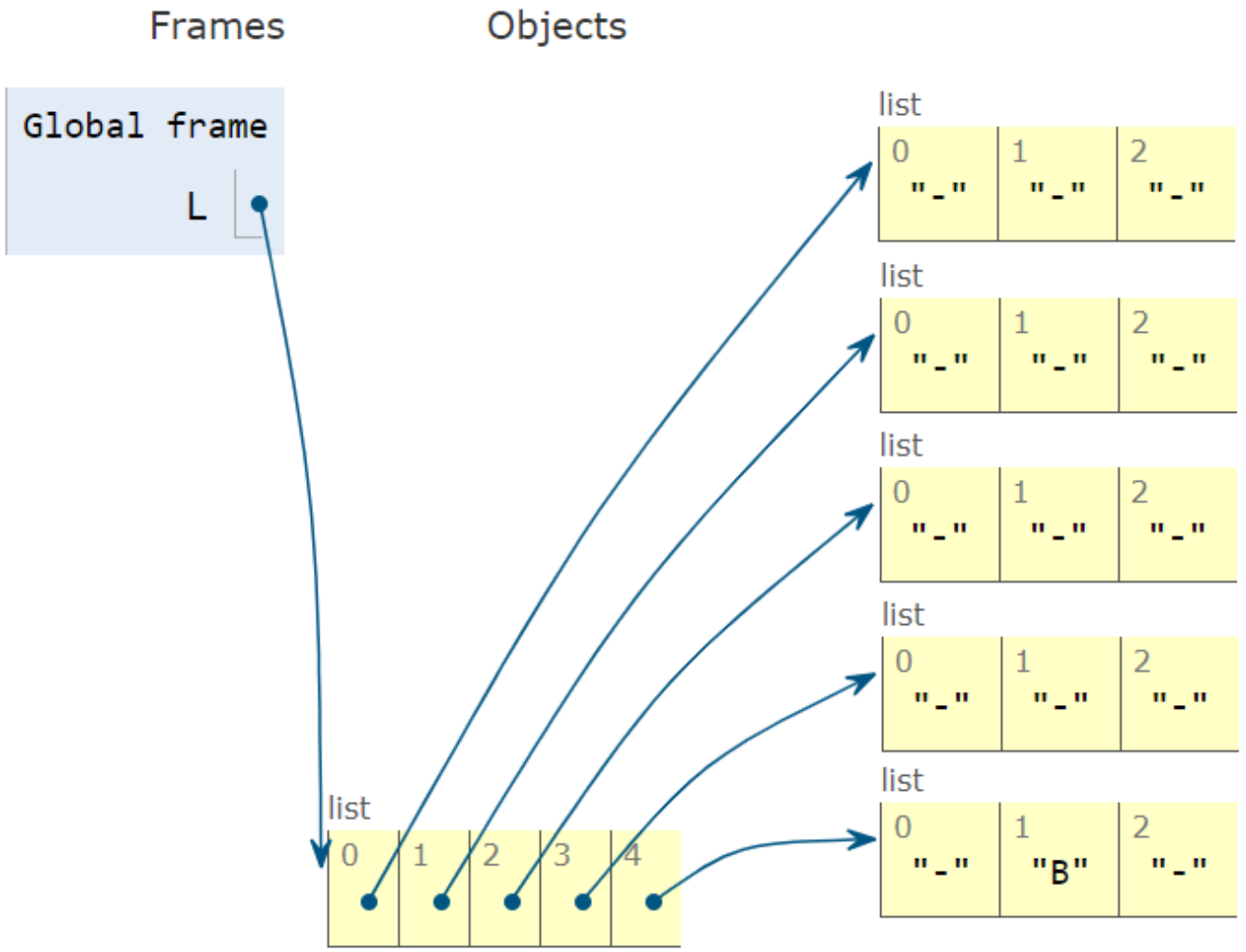
```
blankRow = ['-','-','-']
L = []
L.append(blankRow)
L.append(blankRow)
L.append(blankRow)
L.append(blankRow)
L.append(blankRow)

L[4][1] = 'B'
```



Aliasing

Two variables are “aliases” are when they reference the exact same object

```
L = []
L.append(['-','-','-'])
L.append(['-','-','-'])
L.append(['-','-','-'])
L.append(['-','-','-'])
L.append(['-','-','-'])

L[4][1] = 'B'
```



Poll 8

Which of these is the best code to create a blank word search board?

```
numRows, numCols = 4, 3
```

A.
```
board = []
for r in range(numRows):
    board.append([' ']*numCols)
```

B.
```
board = []
for r in range(numRows):
    cell = [' ']
    row = cell*numCols
    board.append(row)
```

C.
```
cell = [' ']
row = cell*numCols
rowIn2DList = [row]
board = rowIn2DList*numRows
```

D.
```
board = [[' ']*numCols]*numRows
```

Creating 2D Lists

Options to create a "blank" 2D list

```
grid = []
for i in range(numRows):
    grid.append([0]*numCols)
```

Clearly loop through each location

```
grid = []
for i in range(numRows):
    row = []
    for j in range(numCols):
        row.append(0)
    grid.append(row)
```

Fashionable Python: more concise with list comprehension

```
grid = [[0]*numCols for i in range(numRows)]
```

Be carefull!

```
board = [[0]*numCols]*numRows   # Aliased!!
```

Word Search Case Study

Word Search

Twilight

E	I	V	O	L	T	U	R	I	N	E	T
W	D	D	N	W	E	R	E	W	O	L	F
A	V	E	W	P	E	T	C	I	V	L	U
O	A	L	A	L	O	W	L	T	B	S	A
N	G	L	D	F	A	I	I	T	A	P	A
E	A	L	G	T	I	L	P	S	W	A	N
L	S	A	N	E	R	I	S	N	V	T	A
L	E	A	I	D	O	G	E	V	A	K	R
U	A	L	K	W	T	H	W	C	M	C	R
C	A	L	A	A	C	T	E	R	P	A	E
L	O	E	E	R	I	L	S	L	I	L	T
A	A	B	R	D	V	T	V	G	R	B	L
U	C	B	B	N	O	O	M	W	E	N	O
R	B	O	C	A	J	G	O	N	S	A	V

TWILIGHT
SAGA
NEW MOON
ECLIPSE
BREAKING DAWN
BELLA
SWAN
EDWARD
CULLEN
VAMPIRES
WEREWOLF
JACOB
BLACK
VICTORIA
VOLTERRA
VOLTURI

Word Search Top-down Design

Word Search Top-down Design

def wordSearch(grid, word)

For each starting position

If start letter doesn't match

BAIL

For each direction

For each letter in word

If out of bounds

BAIL

If letter doesn't match

BAIL

Word Search Top-down Design

```
def wordSearch(grid, word)
        For each starting position
                If start letter doesn't match
                        return None
                searchFromPos()

def searchFromPos():
        For each direction
                searchFromPosInDir()

def searchFromPosInDir()
        For each letter in word
                if outOfBounds()
                        return None
                if letter doesn't match grid
                        return None
```

Word Search Top-down Design

```
def wordSearch(grid, word):
    gridHeight = len(grid)
    gridWidth = len(grid[0])

    for i in range(gridHeight):
        for j in range(gridWidth):
            if grid[i][j] != word[0]:
                continue
            result = searchFromPos(grid, word, i, j)
            if result is not None:
                return result
    return None
```

Word Search Top-down Design

```
def searchFromPos(grid, word, i, j):
    for dir in getDirections():
        result = searchFromPosInDir(grid, word, i, j, dir)
        if result is not None:
            return result

    return None


def getDirections():
    directions = []
    for i in (-1, 0, 1):
        for j in (-1, 0, 1):
            if i != 0 or j != 0:
                directions.append((i, j))
    return directions
```

(-1,-1)	(-1,0)	(-1,1)
(0,-1)	(0,0)	(0,1)
(1,-1)	(1,0)	(1,1)

Word Search Top-down Design

(-1,-1)	(-1,0)	(-1,1)
(0,-1)	(0,0)	(0,1)
(1,-1)	(1,0)	(1,1)

```
def searchFromPosInDir(grid, word, iStart, jStart, dir):
    gridHeight, gridWidth = len(grid), len(grid[0])
    i, j = iStart, jStart

    # Can skip first position
    i += dir[0]
    j += dir[1]
    for letter in word[1:]:
        if not checkBounds(i, j, gridWidth, gridHeight):
            return None

        if grid[i][j] != letter:
            return None

        i += dir[0]
        j += dir[1]

    return (word, iStart, jStart, dir)
```

```
def checkBounds(i, j, width, height):
    return (0 <= i < height) and (0 <= j < width)
```

E	I	V	O	L	T	U	R	I	N
W	D	D	N	W	E	R	E	W	O
A	V	E	W	P	E	T	C	I	V
O	A	L	A	L	O	W	L	T	B
N	G	L	D	F	A	I	I	T	A
E	A	L	G	T	I	L	P	S	W
L	S	A	N	E	R	I	S	N	V
L	E	A	I	D	O	G	E	V	A