CS100 Python
Introduction to Programming

Lecture 25. Sequence, Set and Mapping

Fu Song

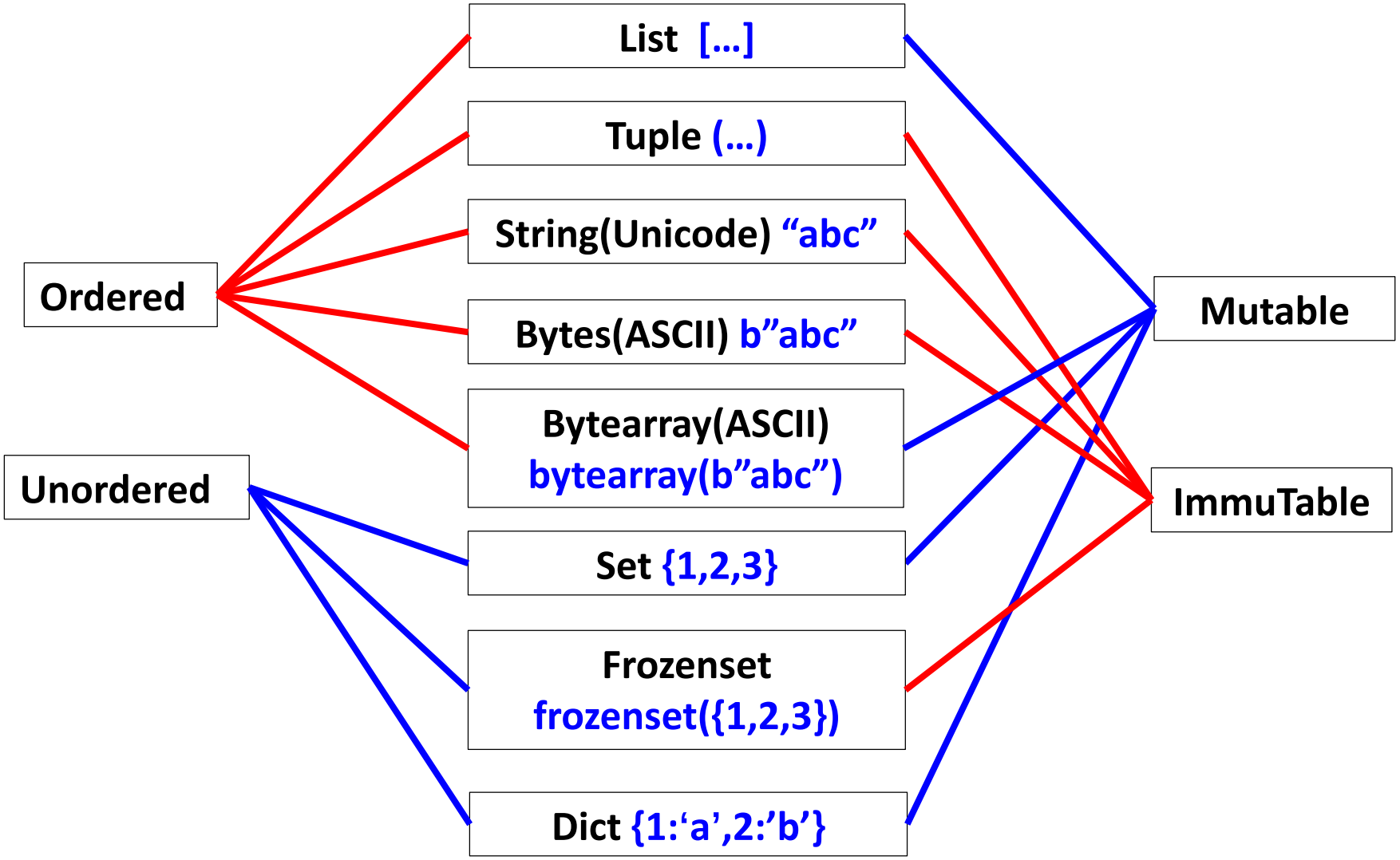
School of Information Science and Technology

ShanghaiTech University

Learning Objectives

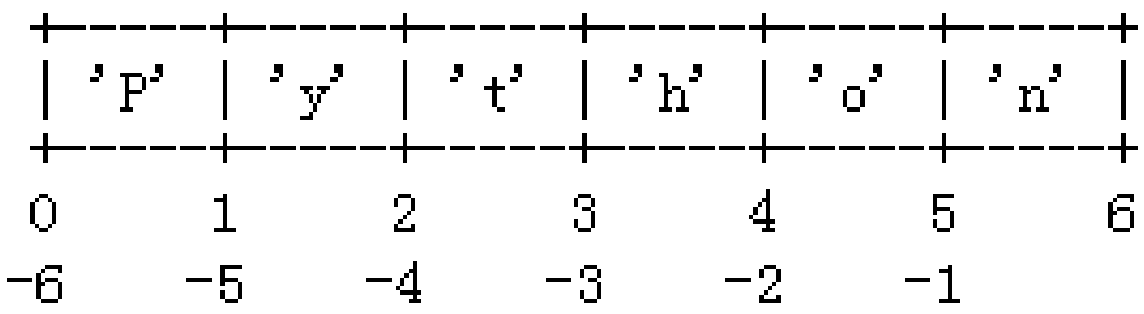
- **Understand and use *compound* data types, used to group together other values**
  - **List**
  - **Tuple**
  - **String**
  - **Bytes and Bytearray**
  - **Set and Frozenset**
  - **Dictionary**

Overview


List […]
Tuple (…)
String(Unicode) “abc”
Bytes(ASCII) b”abc”
Bytearray(ASCII)
bytearray(b”abc”)
Set {1,2,3}
Frozenset
frozenset({1,2,3})
Dict {1:‘a’,2:’b’}
Ordered
Unordered
Mutable
ImmuTable

Element Access

- Elements of the ordered types list, tuple, string, bytes, bytearray and range can be accessed via index
- Python supports bidirectional index
  - First element is indexed by 0
  - The i-th element is indexed by i-1
  - The last element can also accessed via index -1
  - The last i-th element can be accessed via index -i



List

- List is a built-in class in Python
- Typically used to store a sequence of ordered elements in '[' ']' separated by ','

  [1, 'a', 2, 'bc'] [1, 'a', [3, 'b'], 4]
- The types of elements in a list can be distinct,
  - elements can be objects of int, float, complex, list, set, dict, string,……
  - different from array, list, vector of C/C++ whose elements should have same type
- List is mutable, namely, one can add and remove elements from a list 即，也就是

List: Create

- A list can be created by the following ways
  1. x = [] # empty list
  2. x = list() # empty list
  3. x = [1,2,3]
  4. x = [1, 'a', [3, 'b'], 4] # nested list
- A list can be created by transforming an iterable objects, like tuple, range, string via type casting
  1. x = list(range(2,10,3)) # list from range
  2. x = list((2,5,8)) # list from tuple (2,5,8), see later
  3. x = list ("258") # list from string “258”

List: Access

- **__getitem__(self, n):** get the element at the index n of the list object self

  lst[n]

- **__setitem__(self,n,v):** set the element at index n as the object v in the list object self

  lst[n] = v

List: Delete

- The del statement
  - delete whole list

    del lst
  - Which breaks the binding relation between the name and the list object

  - delete an element of a list

    del lst[n]
  - all right elements move to left

List: Example

```
>>> x = [1,'ab',[2,3,4],(1,2)]
>>> x[0]
1
>>> x[-1]
(1, 2)
>>> del x[1]
>>> x
[1, [2, 3, 4], (1, 2)]
>>> del x
>>> x
Traceback (most recent call last):
  File "<pyshell#73>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>>
```

Elements have different types

Access elements via index

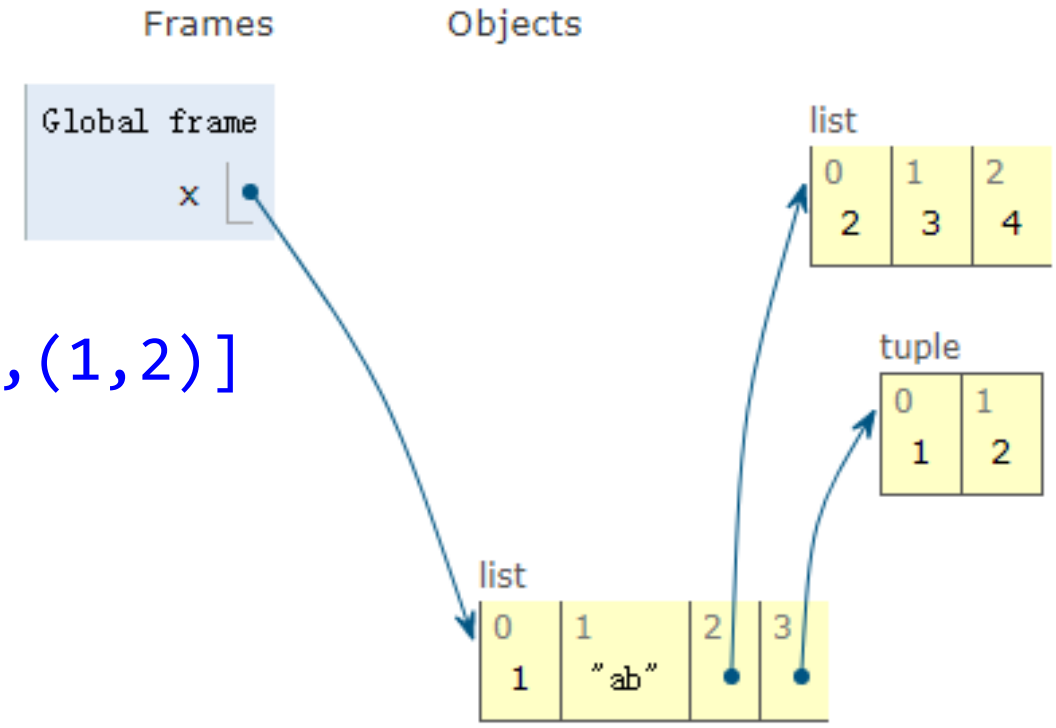
del elements from the list via index

del whole list

List: Insights

- x = […] binds the name x to a list object
- List only stores the reference of elements (objects) in memory, rather than the objects

```
x = [1,'ab',[2,3,4],(1,2)]
```



List: Add elements

1. __add__(self, other): concatenate two lists self and other, return a new object
2. extend(self, other): append all the references of elements in the iterable object other at the end of the self object, return None
3. append(self, e): append the reference of e at the end of the self object
4. __mul__(self, n): duplicates n times of references in self object, return a new object

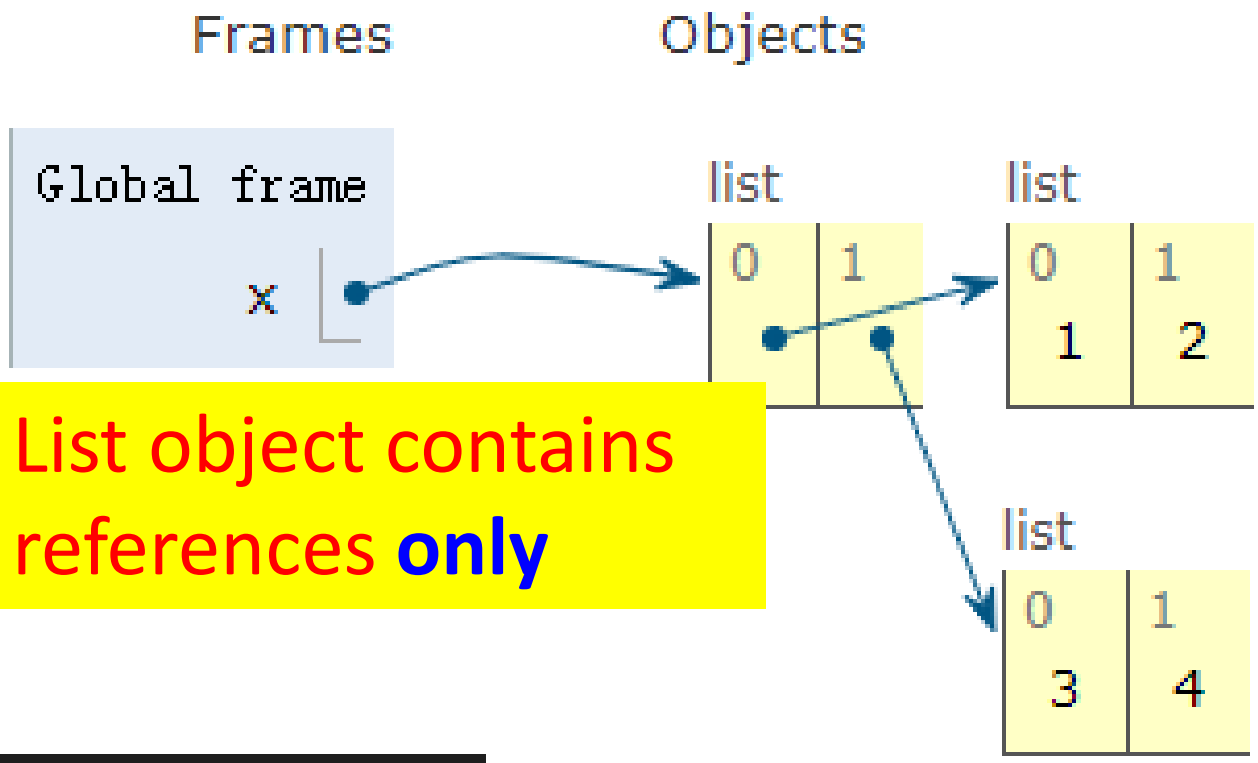
List: __add__

- __add__: is overriding of +

  lst1+lst2

- Create a new list contains all references of elements in lst1 and lst2
  - In the order of their appearance in lst1
  - Then their appearance in lst2
- It only add references of elements into the new list, rather than objects

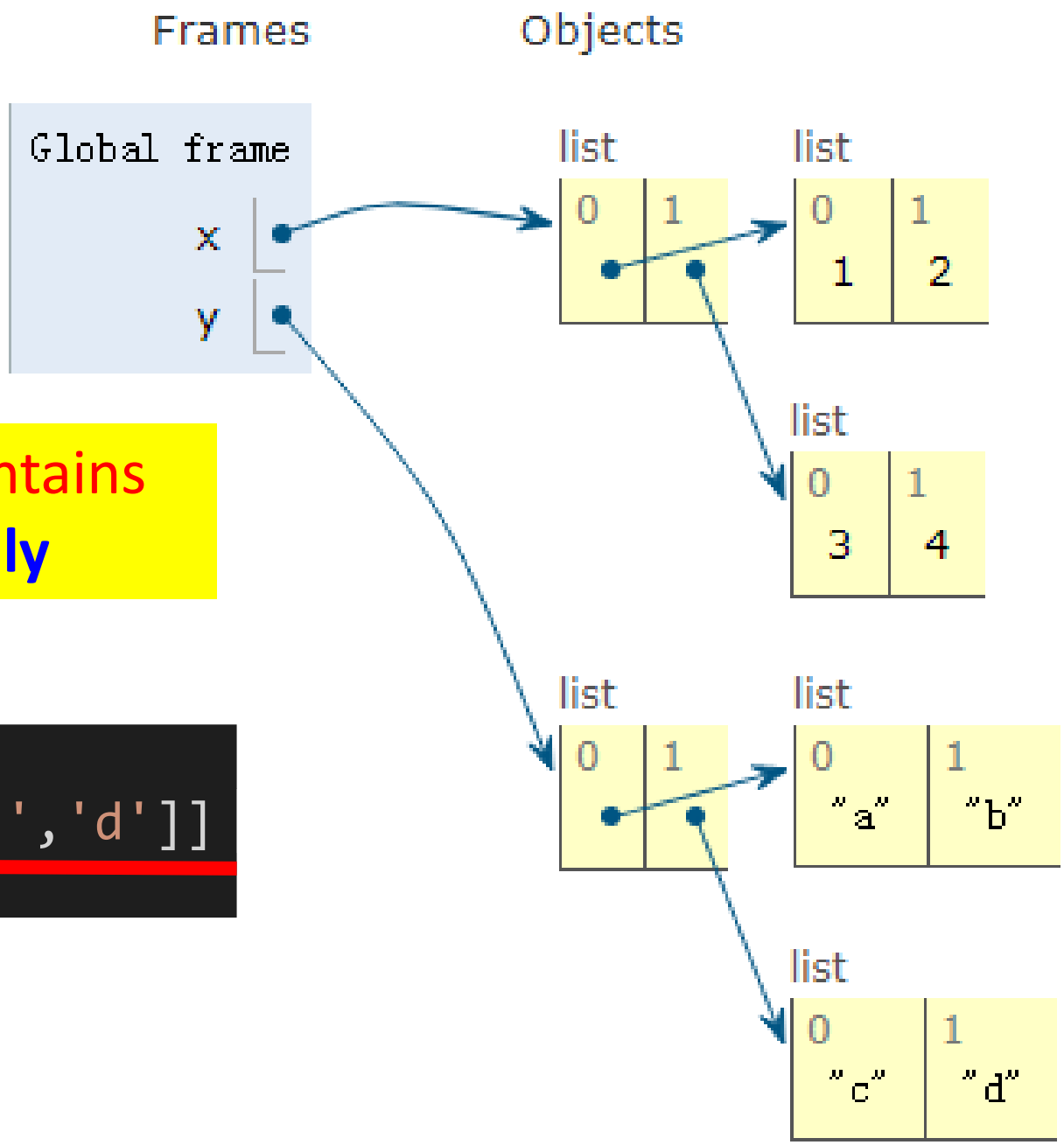
List: Example



List object contains references **only**

```
x = [[1,2],[3,4]]
y = [['a','b'],['c','d']]
z = x+y
```

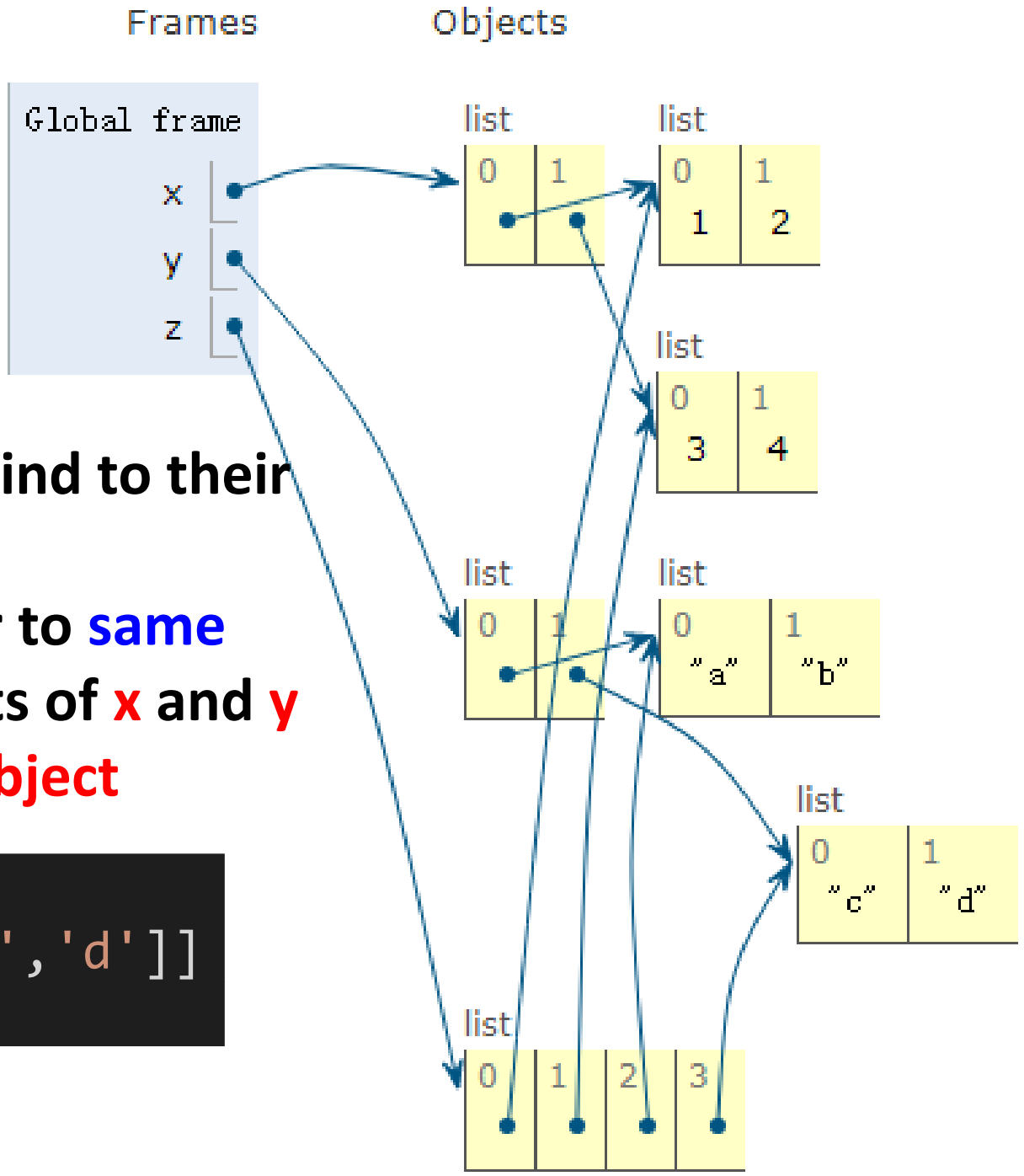
List: Example



List object contains references **only**

```
x = [[1,2],[3,4]]
y = [['a','b'],['c','d']]
z = x+y
```

List: Example



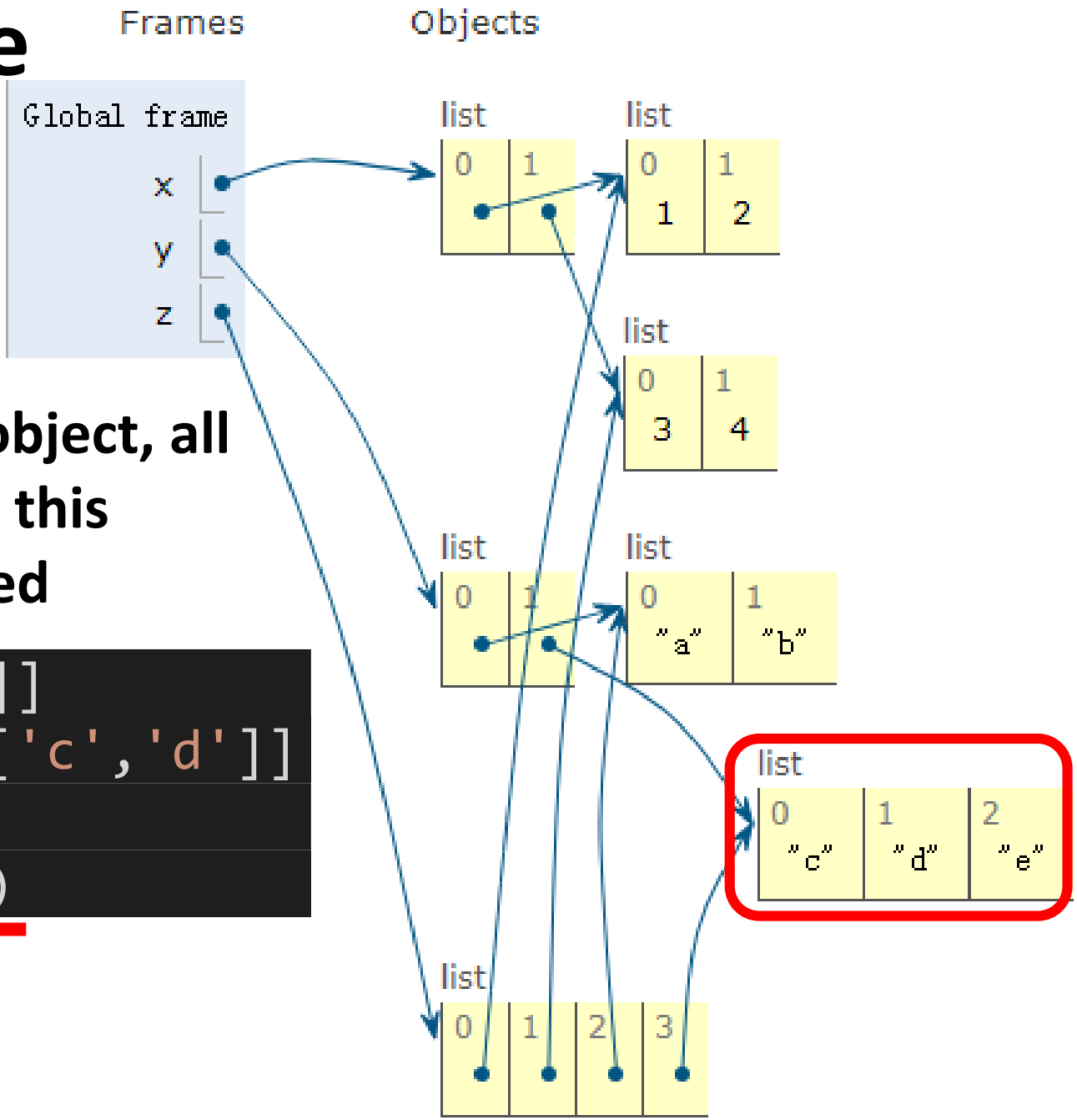
- Names x, y and z bind to their own objects
- Elements of z refer to same objects as elements of x and y
- Z binds to a new object

```
x = [[1,2],[3,4]]
y = [['a','b'],['c','d']]
z = x+y
```

List: Example

- **If we change an object, all the lists referring this object are changed**

```
x = [[1,2],[3,4]]
y = [['a','b'],['c','d']]
z = x+y
y[1].append('e')
```



List: extend

- extend() method

  lst.extend(other)

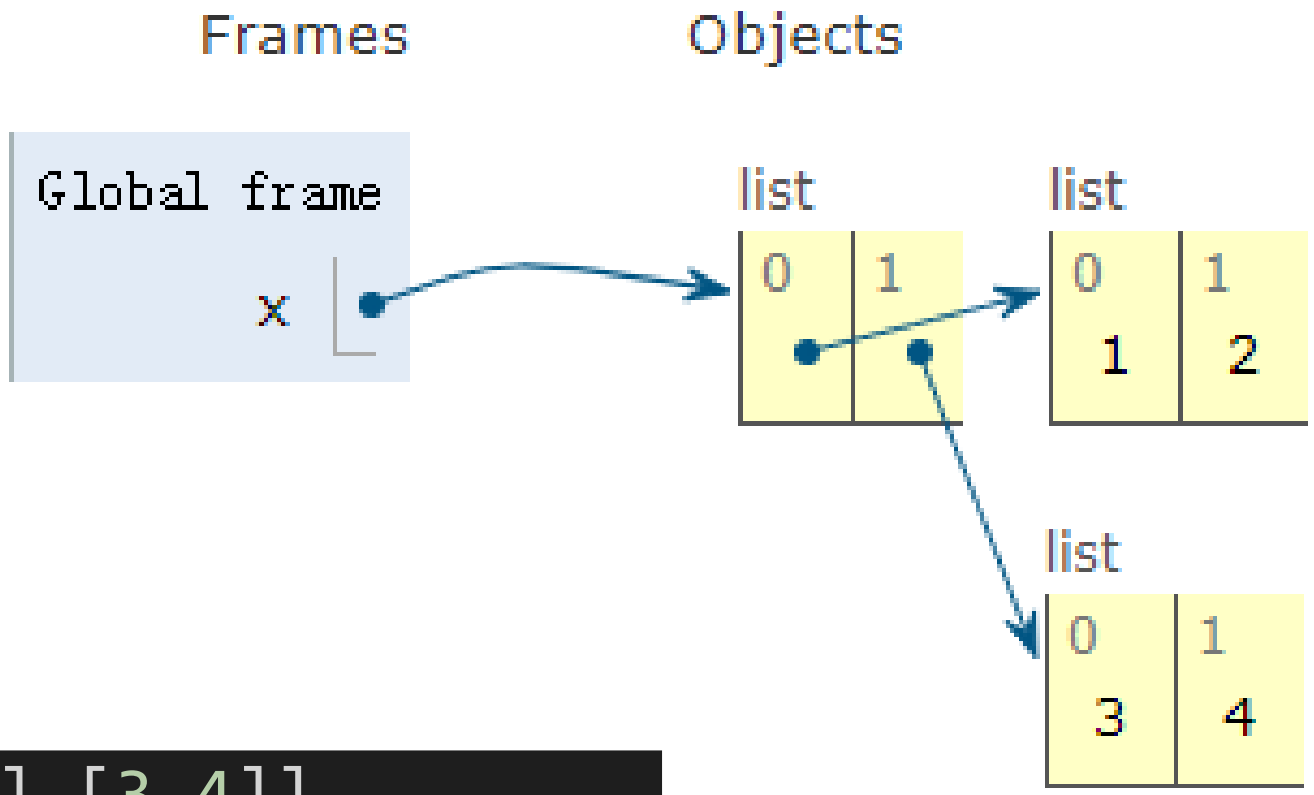
- append all the references of elements in the iterable object other at the end of the list object lst
  - return None rather than the list object lst

List: Example

```
x = [[1,2],[3,4]]
y = (('a','b'),('c','d'))
z = x.extend(y)
```

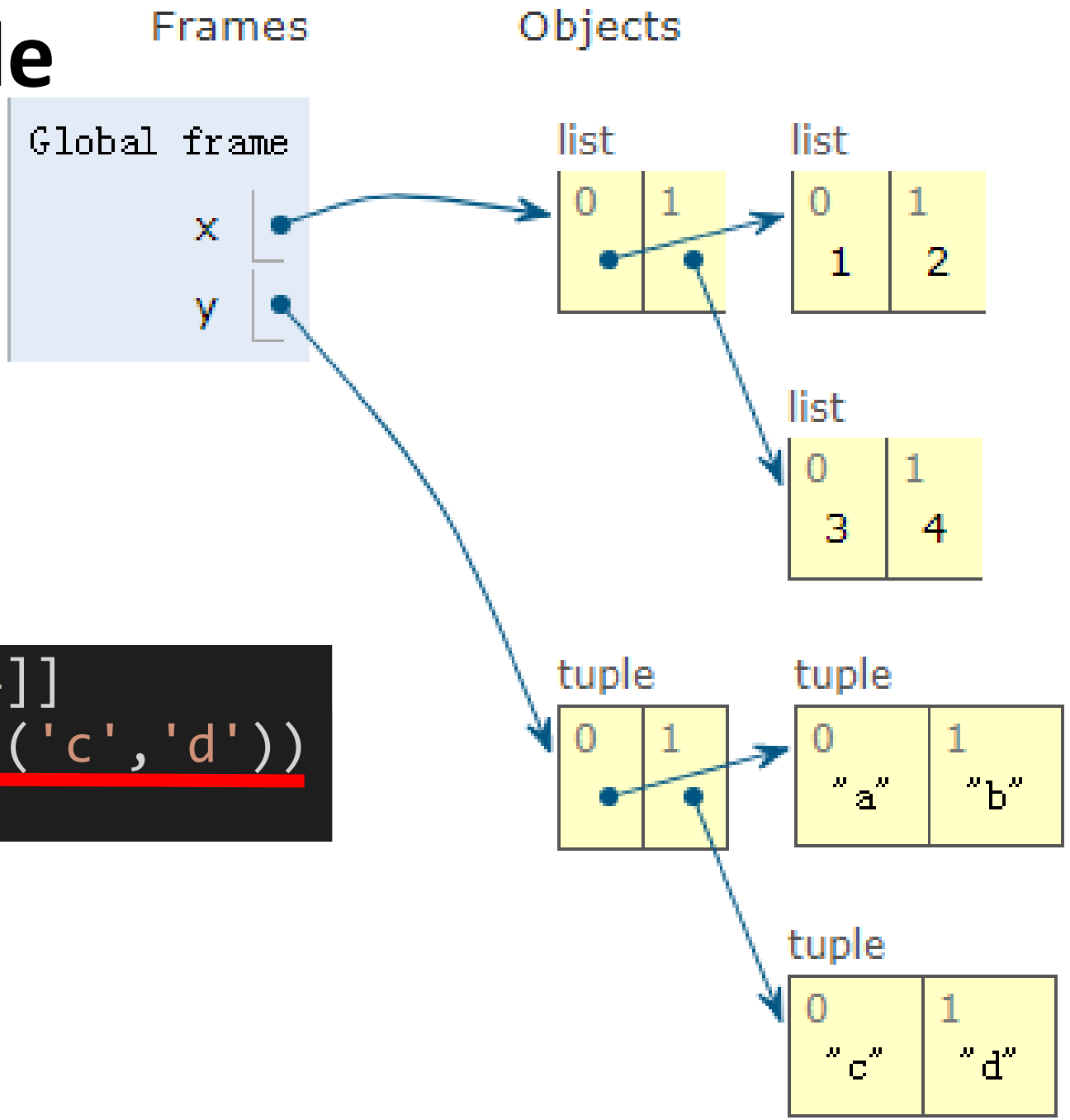
List contains tuples

List: Example



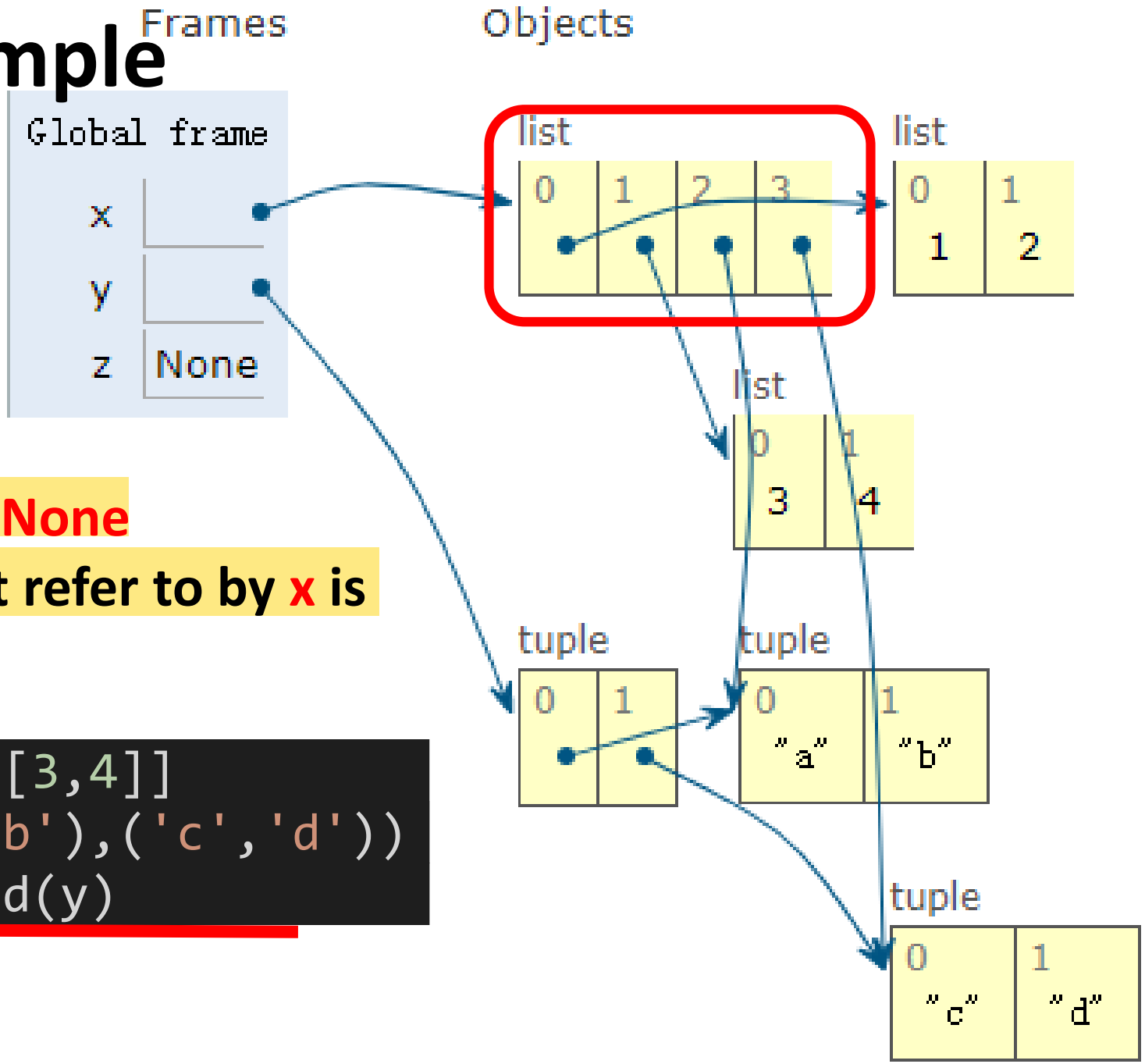

```
x = [[1,2],[3,4]]
y = (('a','b'),('c','d'))
z = x.extend(y)
```

List: Example




```
x = [[1,2],[3,4]]
y = (('a','b'),('c','d'))
z = x.extend(y)
```

List: Example



- **z binds to None**
- **The object refer to by x is updated**

```
x = [[1,2],[3,4]]
y = (('a','b'),('c','d'))
z = x.extend(y)
```

__add__ vs extend

- __add__(self, other):
  - creates a new object, then copies all references of elements in self and other into the new object
- extend(self, other):
  - appends all references of elements in other at the end of self
- extend is faster than __add__ when self and other are large

__add__ vs Extend

```
import time

x = list(range(10000000))
y = list(range(10000000))
start = time.time()
z = x + y
print('__add__:', time.time()-start)

start = time.time()
x.extend(y)
print('extend:', time.time()-start)
```

Output

```
__add__: 0.1930108070373535 2
extend: 0.1480081081390380 9
```

List: append

- append() method

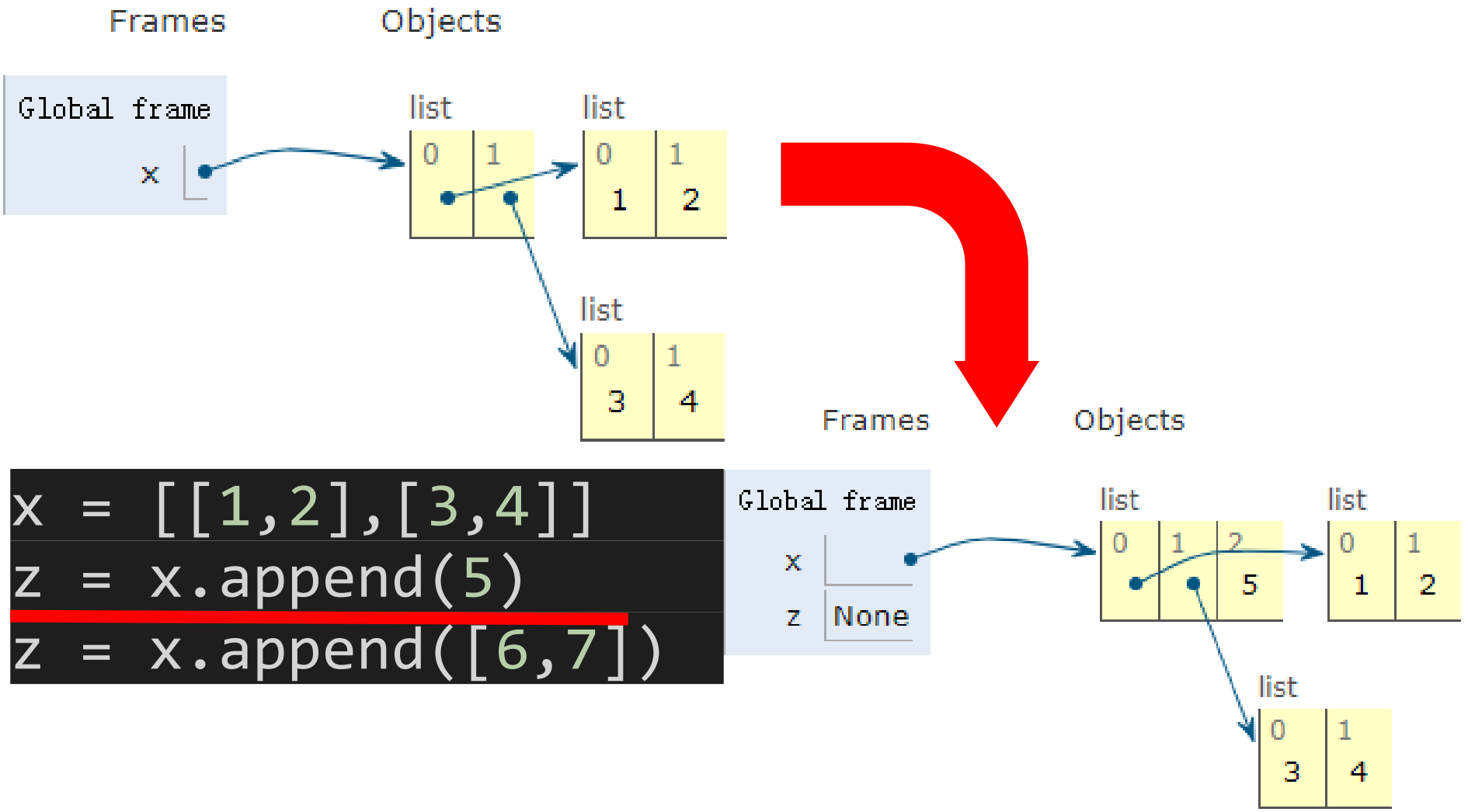
  lst.append(elem)

- append the reference of elem at the end of the list object lst
  - return None rather than the list object lst

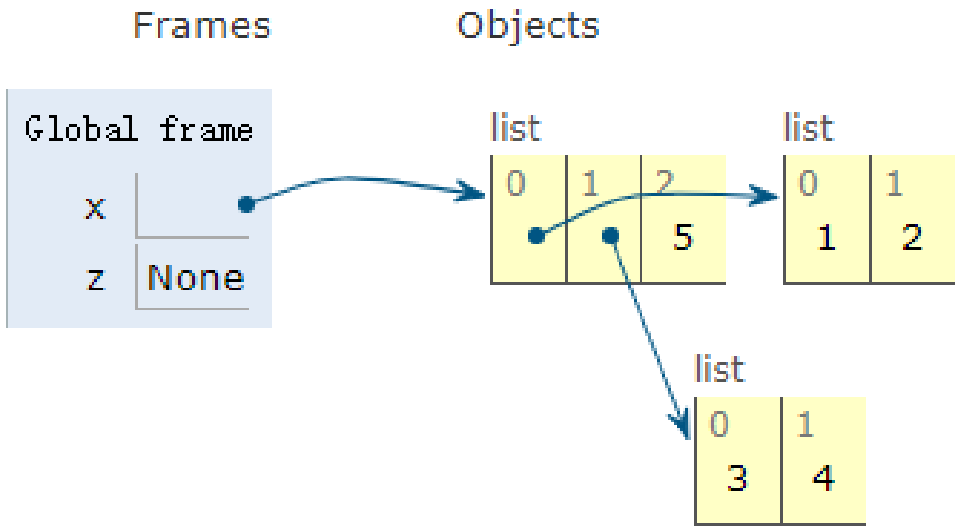
List: Example

```
x = [[1,2],[3,4]]
z = x.append(5)
z = x.append([6,7])
```

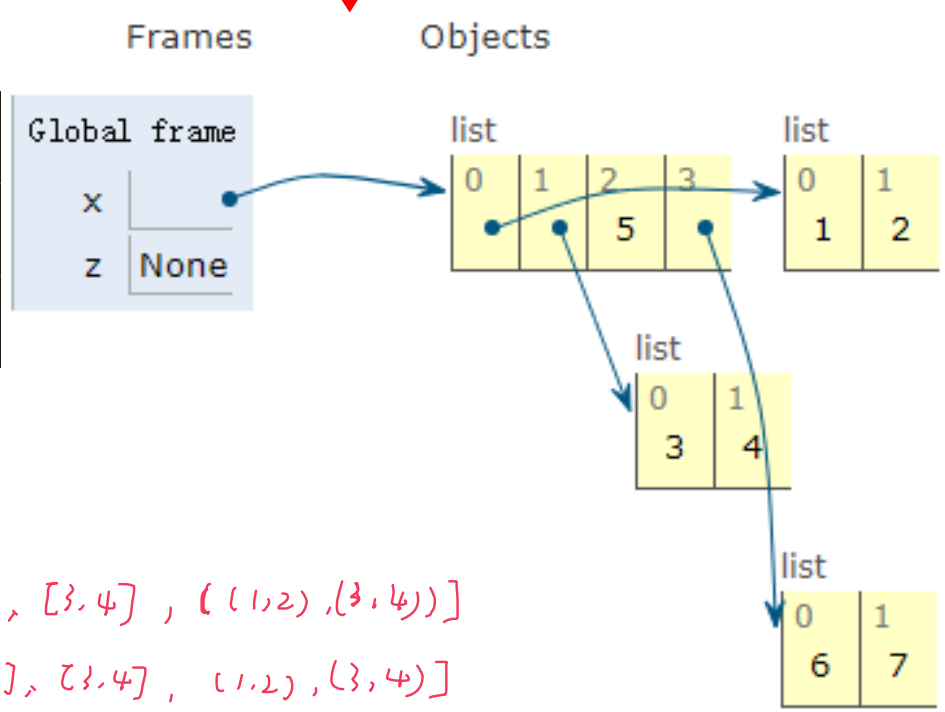
List: Example



List: Example




```
x = [[1,2],[3,4]]
z = x.append(5)
z = x.append([6,7])
```



x = [[1,2] , [3,4]]

y = ((1,2) , (3,4))

x.append(y) ⇒ x = [[1,2] , [3,4] , ((1,2) , (3,4))]

x.extend(y) ⇒ x = [[1,2] , [3,4] , (1,2) , (3,4)]

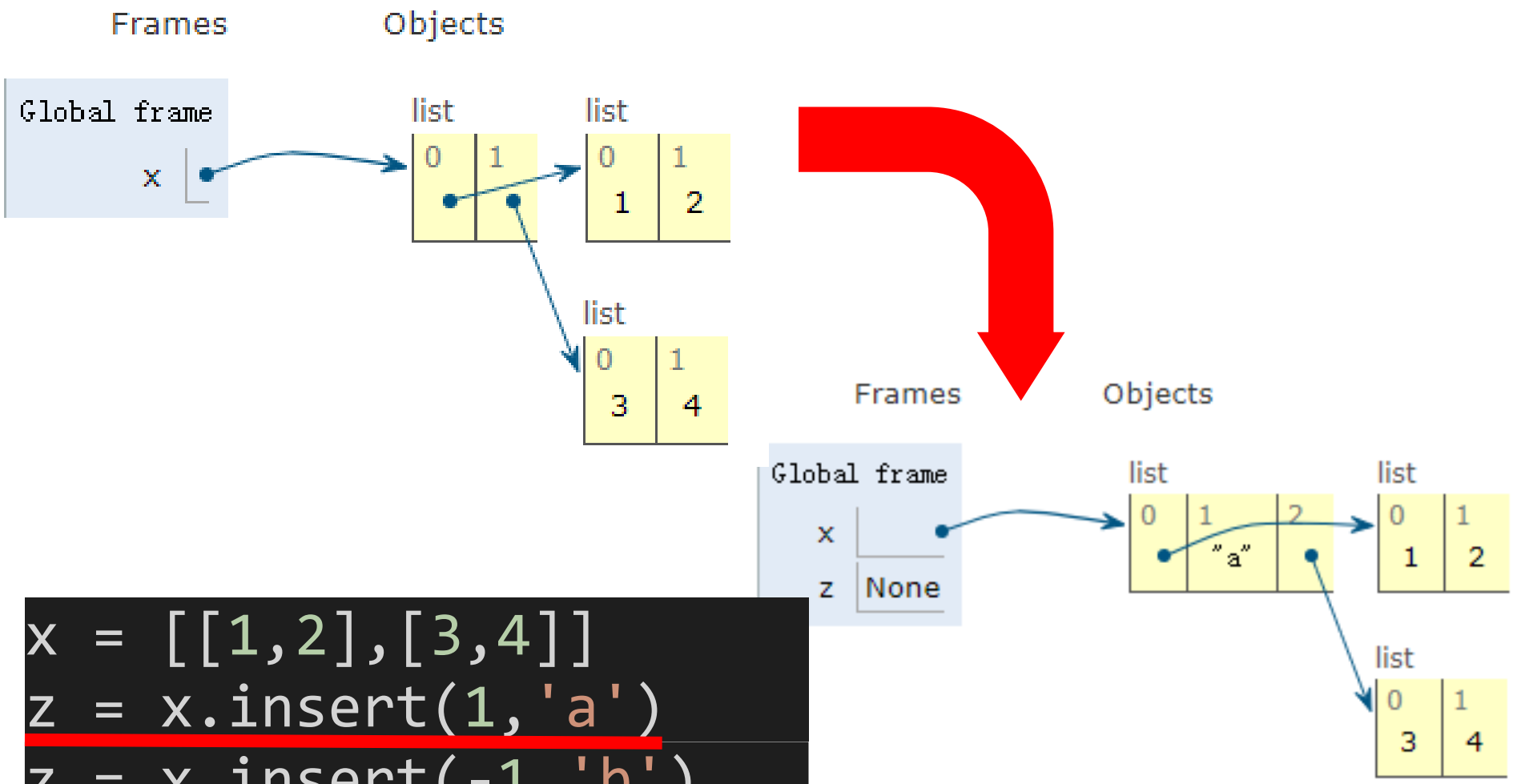
List: insert

- insert(self,i,e)
  - If i is in the bound of the list, insert the reference of e at the index i in the self object
  - Otherwise, append the reference of e at the end of the self object
  - Return None
- It is slower than append, as insert will move all the elements on the right of the position i to right

List: Example

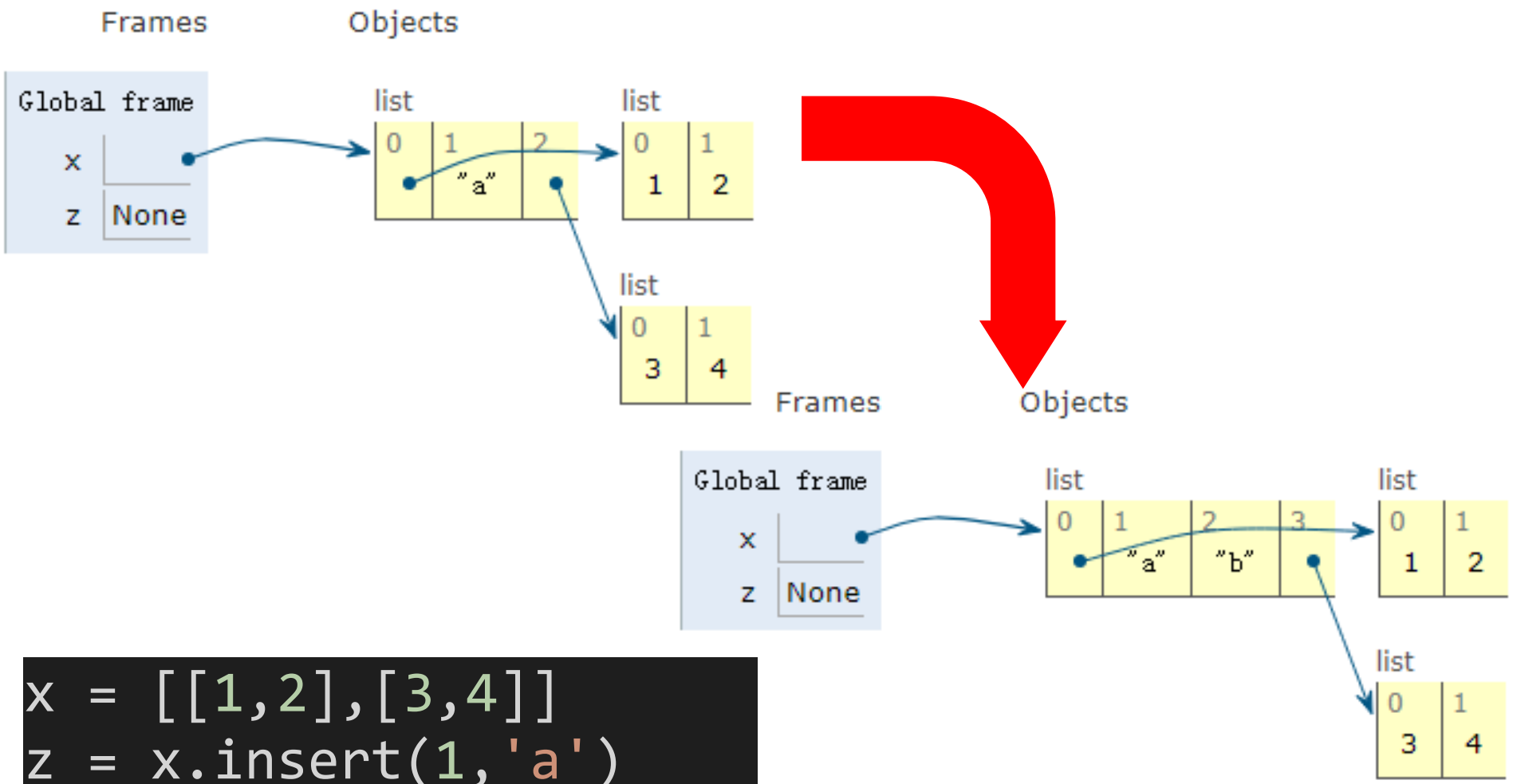
```
x = [[1,2],[3,4]]
z = x.insert(1,'a')
z = x.insert(-1,'b')
z = x.insert(10,'c')
```

List: Example



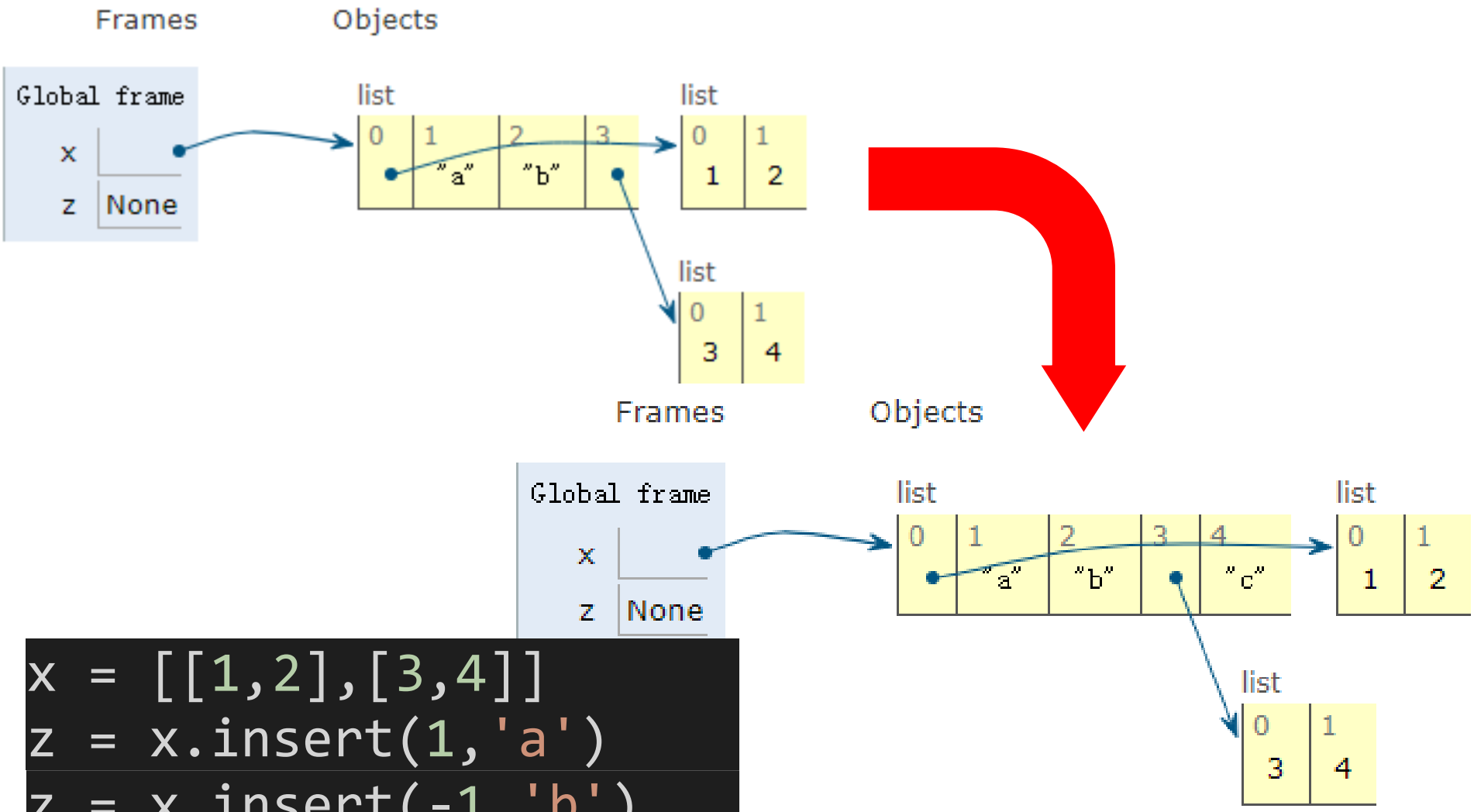

```
x = [[1,2],[3,4]]
z = x.insert(1,'a')
z = x.insert(-1,'b')
z = x.insert(10,'c')
```

List: Example




```
x = [[1,2],[3,4]]
z = x.insert(1,'a')
z = x.insert(-1,'b')
z = x.insert(10,'c')
```

List: Example




```
x = [[1,2],[3,4]]
z = x.insert(1,'a')
z = x.insert(-1,'b')
z = x.insert(10,'c')
```

append vs insert

```
import time
start = time.time()
x = []
for i in range(100000):
    x.insert(0,i)
print('Insert:', time.time()-start)

y =[]
start = time.time()
for i in range(100000):
    y.append(i)
print('Append:', time.time()-start)
```

Output

Insert: 2.6971559524536133
Append: 0.018001079559326172

List: __mul__

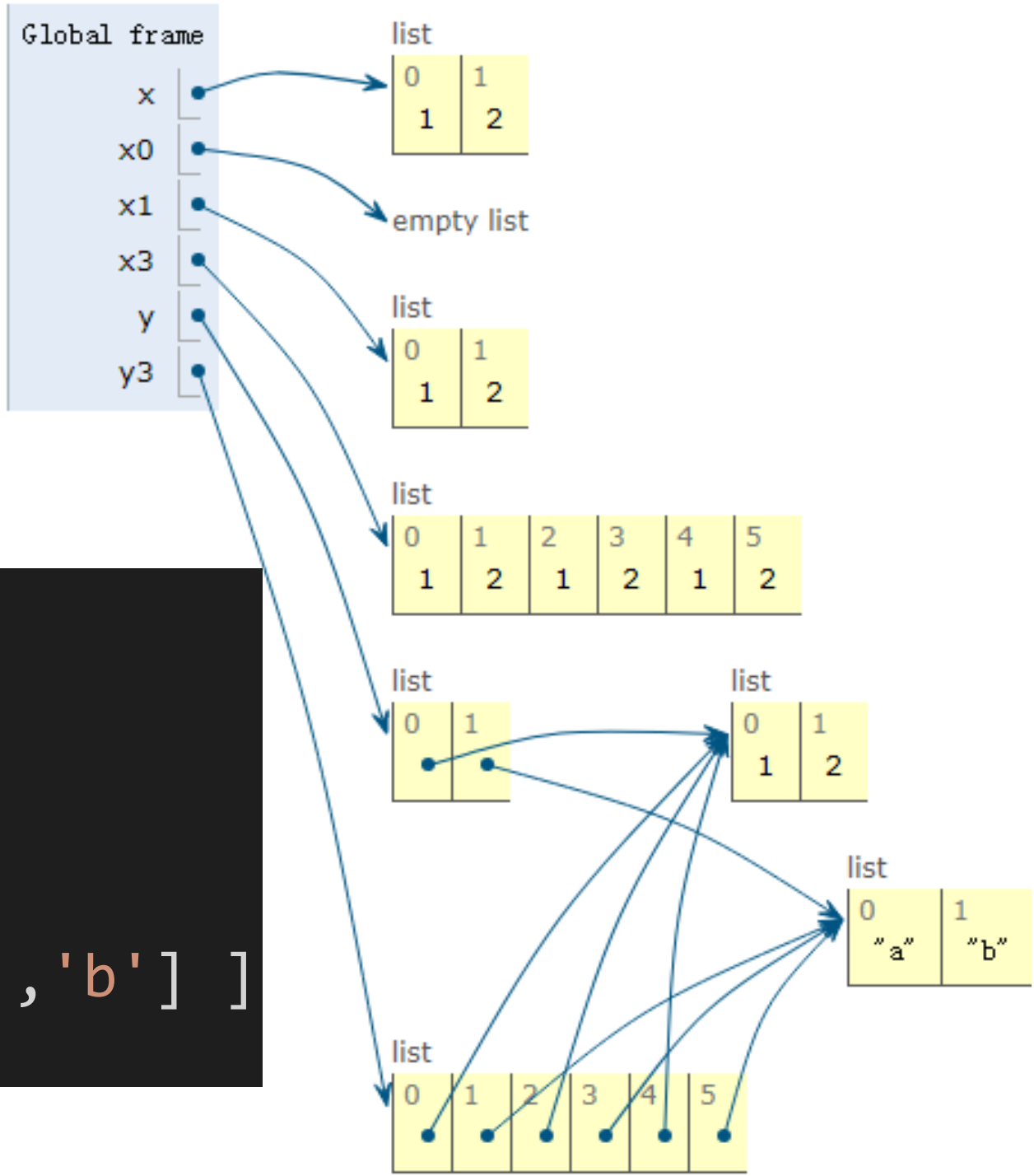
- __mul__ (self,n): is overriding of *
  - n must be an integer, otherwise TypeError
  - If n <=0, then return an empty list []
  - If n ==1, then return a new list contains the same references of the object self
  - Otherwise, return a new list contains n times of references of the object self

List: Example

```
x = [1,2]
x0 = x * 0
x1 = x * 1
x3 = x * 3

y = [ [1,2],['a','b'] ]
y3 = y * 3
```

List: Example

```
x = [1,2]
x0 = x * 0
x1 = x * 1
x3 = x * 3

y = [ [1,2],['a','b'] ]
y3 = y * 3
```



List: Delete elements

1. del lst[n] i.e., __delitem__(self,n) : delete element at index n
2. pop(self, n=-1): delete and return the element at index n (in default, the last one)
3. remove(self, e): delete the first element e' such that e==e'

- All the right elements move to left one step
- IndexError (n out of the bound of the list)
- ValueError (e does not appear in the list)

List: Example

```
x = [1,2,'a','b',[3,4],[5,6]]
del x[1]
print(x)
y = x.pop()
print(y)
print(x)
y = x.pop(1)
print(y)
x.remove('b')
print(x)
```

Output

```
[1, 'a', 'b', [3, 4], [5, 6]]
[5, 6]
[1, 'a', 'b', [3, 4]]
a
[1, [3, 4]]
>>>
```

List: Common operations

```
__add__(self, value, /)
    Return self+value.

__mul__(self, value, /)
    Return self*value.

__rmul__(self, value, /)
    Return value*self.

__iadd__(self, value, /)
    Implement self+=value.

__imul__(self, value, /)
    Implement self*=value.
```

List: Common operations

```
__contains__(self, key, /)
    Return key in self.
__eq__(self, value, /)
    Return self==value.

__ge__(self, value, /)
    Return self>=value.

__gt__(self, value, /)
    Return self>value.

__le__(self, value, /)
    Return self<=value.

__lt__(self, value, /)
    Return self<value.

__ne__(self, value, /)
    Return self!=value.
```

Lexicographical ordering

Lexicographical ordering

1. the first two items are compared,
   - if differ, determines the result;
   - if equal, next two items are compared, and so on, until either sequence is exhausted.
2. If two items to be compared are themselves sequences of the same type, lexicographical comparison is carried out recursively
3. If all items of two sequences compare equal, then two sequences are equal
4. If one sequence is an initial sub-sequence of the other, the shorter one is the smaller one
5. raise a TypeError exception if two items are incomparable

```
__delitem__(self, key, /)
    Delete self[key].

__getattribute__(self, name, /)
    Return getattr(self, name).

__getitem__(...)
    x.__getitem__(y) <==> x[y]

__setitem__(self, key, value, /)
    Set self[key] to value.

__len__(self, /)
    Return len(self).

__sizeof__(self, /)
    Return the size of the list in memory, in bytes.

__repr__(self, /)
    Return repr(self).

__reversed__(self, /)
    Return a reverse iterator over the list.
```

Get **class** attribute

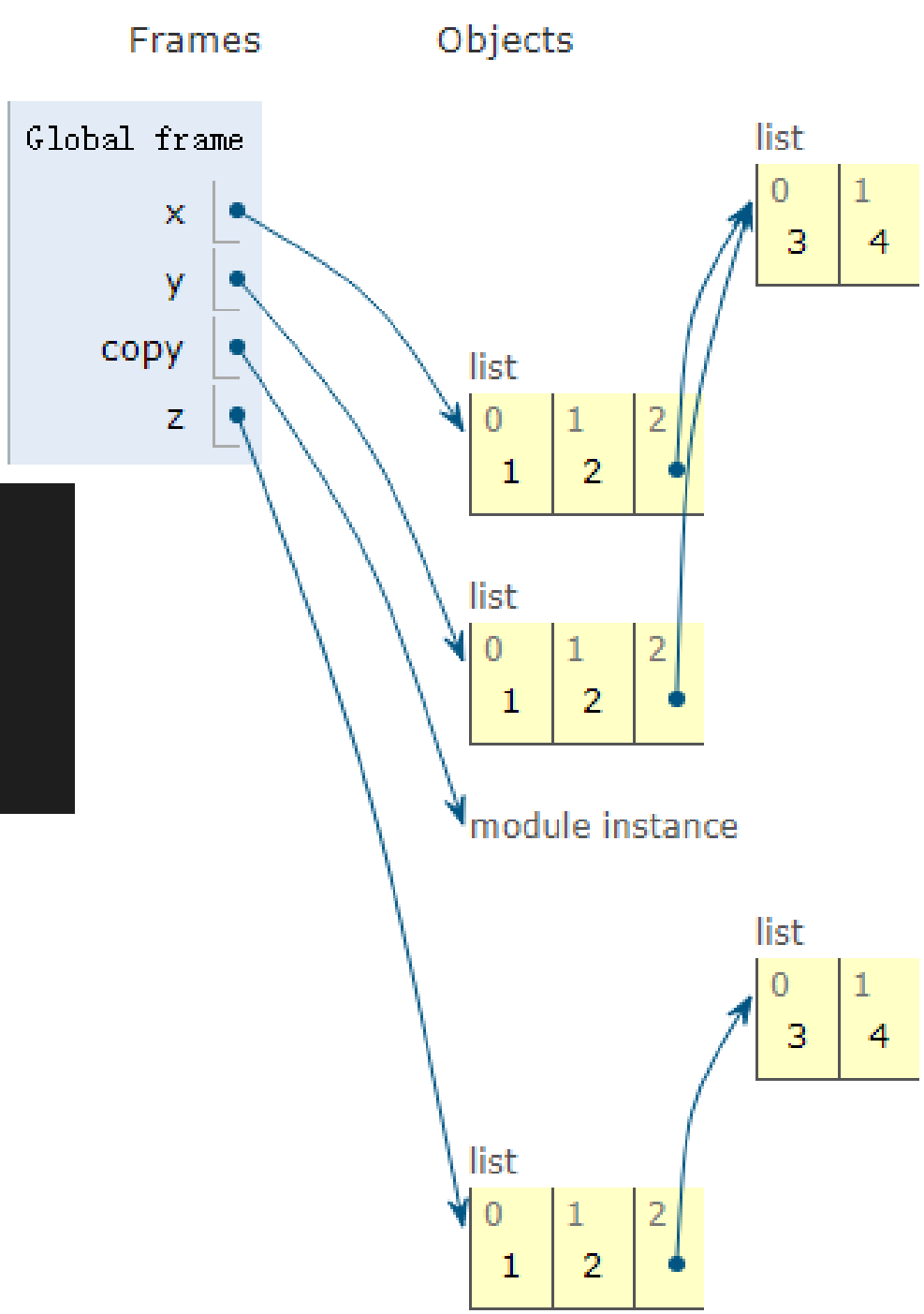
Raise **IndexError** if the index is out of bounds

List: Common operations

```
>>> x =[1,2,[3,4]]
>>> len(x)
3
>>> x.__sizeof__()
32
>>> x.__repr__()
'[1, 2, [3, 4]]'
>>> print(x.__reversed__())
<list_reverseiterator object at
0x02BBA330>
>>>
```

```
append(self, object, /)
    Append object to the end of the list.

clear(self, /)
    Remove all items from list.

copy(self, /)
    Return a shallow copy of the list.

count(self, value, /)
    Return number of occurrences of value.

extend(self, iterable, /)
    Extend list by appending elements from the iterable.

index(self, value, start=0, stop=2147483647, /)
    Return first index of value.

    Raises ValueError if the value is not present.
```

```
x = [1,2,[3,4]]
y = x.copy()
import copy
z = copy.deepcopy(x)
```



```
insert(self, index, object, /)
    Insert object before index.

pop(self, index=-1, /)
    Remove and return item at index (default last).

    Raises IndexError if list is empty or index is out of
range.

remove(self, value, /)
    Remove first occurrence of value.

    Raises ValueError if the value is not present.

reverse(self, /)
    Reverse *IN PLACE*.

sort(self, /, *, key=None, reverse=False)
    Stable sort *IN PLACE*.
```

Lexicographical ordering

List: Common operations

```
>>> x = [3,1,2,4]
>>> x.reverse()
>>> x
[4, 2, 1, 3]
>>> x.sort()
>>> x
[1, 2, 3, 4]
>>> x = [3,1,2,4]
>>> sorted(x)
[1, 2, 3, 4]
>>> x
[3, 1, 2, 4]
```

No new list is created, the list x is changed

注意 sort() 与 sorted() 区别

A new list is created, the list x is unchanged

Warning

```
>>> x = [1,2,1,2,1,2,1,2,1]
>>> for i in x:
    if i == 1:
        x.remove(i)
>>> x
[2, 2, 2, 2]

>>> x = [1,2,1,2,1,1,1]
>>> for i in x:
    if i == 1:
            x.remove(i)
>>> x
[2, 2, 1]
```

Don't change a list when iterate elements of the list using for loop

When 3rd 1 is removed, the 4th 1 is moved to left, loop will miss this 1 and go to the 5th 1

Slicing

lst(start=0: [stop=-1[: step=1]])

```
>>> x = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> x[::]                                  # create a new
list
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> x[::-1]                                # reorder
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
>>> x[::2]                                # even positions
[3, 5, 7, 11, 15]
>>> x[1::2]                                # odd positions
[4, 6, 9, 13, 17]
```

Slicing

lst(start=0: [stop=-1[: step=1]])

```
>>> x = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> x[3::]                          # start from index 3
[6, 7, 9, 11, 13, 15, 17]
>>> x[3:6]                          # index between [3, 6)
[6, 7, 9]
>>> x[0:100:1]                       # the first 100 elements
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> x[100:]                        # starting from index 100
[]
>>> x[100]
IndexError: list index out of range
```

Slicing

lst(start=0: [stop=-1[: step=1]]) = other

```
>>> aList = [3, 5, 7]
>>> aList[len(aList):] = [9]    # append at end
>>> aList
[3, 5, 7, 9]
>>> aList[:3] = [1,2,3] # replace first 3 elements
>>> aList
[1, 2, 3, 9]
>>> aList[:3] = []     #delete the first 3 elements
>>> aList
[9]
```

Slicing

lst(start=0: [stop=-1[: step=1]]) = other

```
>>> aList = list(range(10))
>>> aList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> aList[::2] = [0]*5        #replace even places
>>> aList
[0, 1, 0, 3, 0, 5, 0, 7, 0, 9]
>>> aList[::2] = [0]*3  #number of elements
                        # should same
ValueError: attempt to assign sequence of size 3 to
  extended slice of size 5
```

List comprehensions

- List comprehensions provide a concise way to create lists
- Common applications are:
  1. to make new lists where each element is the result of some operations applied to each member of another sequence or iterable,
  2. or to create a subsequence of those elements that satisfy a certain condition

```
>>> s = [x**2 for x in range(10)]
>>> s
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
>>>
```

Learning Objectives

- **Understand and use *compound* data types, used to group together other values**
  - **List**
  - **Tuple**
  - **String**
  - **Bytes**
  - **Bytearray**
  - **Set and Frozenset**
  - **Dictionaries**

Tuple

- Tuple is a built-in class in Python
- Typically used to store a sequence of ordered elements in '(' ')' separated by ','

  (1, 'a', [3, 'b'], 4)

- Tuple is immutable, namely, one cannot add and remove elements from a tuple (in contrast to List)
- The types of elements in a tuple can be distinct,
  - elements can be objects of int, float, complex, list, set, dict, string,……
  - different from array, list, vector of C/C++ whose elements should have same type

Tuple: Create and Delete

- A tuple can be created by the following ways
  1. x = (1, 'a', [3, 'b'], 4)
  2. x = () or x= tuple() # empty tuple
  3. x =(3,) # tuple has one element, x = (3) assigns 3 to x
- A tuple can be created by transforming an iterable objects, like list, range, string via type casting
  1. x = tuple(range(2,10,3)) # tuple from range
  2. x = tuple([2,5,8]) # tuple from list (2,5,8)
  3. x = tuple("258") # tuple from string “258”
- del statement can be used to delete a tuple, but not elements of a tuple (immutable)

Tuple vs List

- Tuple is immutable, list is mutable
- Tuple support almost all methods of list, excluding
  - append()、extend()、insert() etc. to add elements into a tuple
  - remove()、pop()，del etc. to remove elements from a tuple
- Tuples can be elements of sets and keys of dicts, while list cannot
- Tuple( ) freezes a list, list( ) unfreezes a tuple
- Tuple is more efficient (time, memory, safe) than list
  - Use tuple if only need to store, traverse and read values
- help(tuple) lists all methods of tuple

```
>>> r = range(20)
>>> l = list(r)
>>> t = tuple(r)
>>> r
range(0, 20)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19]
>>> t
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19)
>>> r.__sizeof__()
24
>>> l.__sizeof__()
132
>>> t.__sizeof__()
92
```

Learning Objectives

- **Understand and use *compound* data types, used to group together other values**
  - **List**
  - **Tuple**
  - **String**
  - **Bytes**
  - **Bytearray**
  - **Set and Frozenset**
  - **Dictionaries**

String

- Strings or str objects are immutable sequences of Unicode code points, rather than ASCII
- String literals are written in a variety of ways:

  Single quotes: 'allows embedded "double" quotes'

  Double quotes: "allows embedded 'single' quotes"

  Triple quoted: '''Three single quotes''', """Three double quotes"""
- All the objects that have __str__ instance method can be transformed into a string via str()

ASCII

- One byte for one character, hence total 256 characters

	0	1	2	3	4	5	6	7	8	9
0	NUL							BEL	BS	TAB
1	LF		FF	CR						
2								ESC		
3			SP	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

Unicode

- Unicode can be implemented by different character encodings such as UTF-8, UTF-16, and UTF-32, and several other encodings are in use
- UTF-8, used in over 92% of websites, uses one byte for the first 128 code points(ASCII characters), and up to 4 bytes for other
- Python 3 provides Chinese characters, in default UTF-8
- Note one number、English character or Chinese character is considered as one byte in Python

Unicode: Example

```
>>> x = "宋富".encode("utf8")
>>> x
b'\xe5\xae\x8b\xe5\xaf\x8c'
>>> y = "宋富".encode("utf16")
>>> y
b'\xff\xfe\x8b[\xcc['
>>> x.decode("utf16")
'孻\ue58b貯'
>>>
```

String

- String provides most of common methods for list and tuple including slicing, and many string-specific methods
  - Format string

Format String

- Returns a string by replacing all RF_i's with a_i's with respect formats in RF_i's

"$s_{11}RF_1s_{12}....s_{n1}RF_ns_{n2}$".format($a_1$,…,$a_n$)

```
replacement_field ::=  "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name        ::=  arg_name ("." attribute_name | "[" element_index "]")*
arg_name          ::=  [identifier | digit+]
attribute_name    ::=  identifier
element_index     ::=  digit+ | index_string
index_string      ::=  <any source character except "]"> +
conversion        ::=  "r" | "s" | "a"
format_spec       ::=  <see below>
```

'!s': calls str()
'!r': calls repr()
'!a' calls ascii()

X = "My name is {0} {1}".format("Fu", "Song")

Format String

```
'{0},{1},{2}'.format(x, y, z)
 # argument name
'{},{},{}'.format(x, y, z))
 # default argument, 0,1,2,… can be omitted
'{A}, {B}, {C}'.format(A=x, B=y, C=z))
 # keyword arguments
'{0[1]},{1},{2}'.format(x, y, x))
 # argument with element index
'{0.attr},{1},{2}'.format(x, y, x))
 # argument with attribute name
```

Format Specification

[[fill]align][sign][#][0][width][,|-][.precision][type]

- **fill:** can be any character (default to a space)
- **align:**
  - ‘<‘ left-aligned(default)
  - ’>‘:right-aligned
  - '=‘: fill after sign (only valid for numeric types)
  - '^‘: centered
- **sign:** only valid for number types
  - '+‘: a sign for both positive and negative numbers
  - '-‘: a sign only for negative numbers (default)
  - space: space for positive numbers, a sign for negative numbers
- **#:** '0b', '0o', or '0x‘ for binary, octal, or hexadecimal
- **,|-:** thousands separator
- **0** : sign-aware zero-padding for numeric types

Format Specification: Type

Type	Meaning
's'	String format (default)
'b'	Binary format. Outputs the number in base 2
'c'	Character. Converts the integer to the corresponding unicode character before printing
'd‘	Decimal Integer. Outputs the number in base 10 (default)
'o'	Octal format. Outputs the number in base 8
'x'	Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9
'X'	Hex format. Outputs the number in base 16, using upper-case letters for the digits above 9
'n'	Number. This is the same as 'd', except that it uses the current locale setting to insert the appropriate number separator characters

Format Specification: Example

```
>>> print("{0:,} in hex:{0:#x}, {1} in
oct:{1:#o}".format(5555,55))
5,555 in hex:0x15b3, 55 in oct:0o67

>>> print("{1:,} in hex:{1:#x}, {0} in
oct:{0:#o}".format(5555,55))
55 in hex:0x37, 5555 in oct:0o12663

>>>position = (5, 8, 13)
>>> Print("X:{0[0]};Y:{0[1]};Z:{0[2]}".format(position))
X:5;Y:8;Z:13

>>> print("{0:->+#015,.3f}".format(5555.1))
-----+5,555.100
```

String

- **String-specific methods (search)**
  - **s.find(sub[, start[, end]])** return the lowest index in S where substring sub is found such that sub is contained within s[start:end]; Return -1 on failure
  - **s.rfind(sub[, start[, end]])** return the highest index in S where substring sub is found such that sub is contained within s[start:end]; Return -1 on failure
  - **s.index(sub[, start[, end]])** and **s.rindex(sub[, start[, end]])** are similar to s.find and s.rfind, except that Raises ValueError on failure
  - **s.count(sub[, start[, end]])** return the number of non-overlapping occurrences of substring sub in string s[start:end]

```
>>> s = "apple,peach,banana,peach,pear"
>>> s.find("peach")          # 6
>>> s.find("peach",7)        # 19
>>> s.find("peach",7,20)     # -1
>>> s.rfind('p')             # 25
>>> s.index('p')             # 1
>>> s.index('pe')            # 6
>>> s.index('pear')          # 25
>>> s.index(‘ppp’)     # ValueError: not found
>>> s.count('p')             # 5
>>> s.count('pp')            # 1
>>> s.count('ppp')           # 0
>>> x = "pppp"
>>> x.count("pp")            # 2
```

String

- **String-specific methods (split and partition)**
  - **s.split(self, sep=None, maxsplit=-1)** return a list of the of words (max. No. **maxsplit, -1:no limit**) in the string, using sep as the delimiter string from left to right, seps are excluding in the list; sep=None removes all whitespaces and splits by space
  - **s.rsplit(self, sep=None, maxsplit=-1)** similar to **s.split,** but right to left
  - **s.partition(self, sep)** return a 3-tuple t of words in the string, using sep as the delimiter string from left to right;

    t[0]=substring before sep, t[1]=sep, t[2]=substring after sep

    t[0]=s, t[1]=t[2]=[] if s does not contain sep
  - **s.rpartition (self, sep)** similar to **s.partition** but right to left

```
>>> s = "apple,peach,banana,pear"
>>> s.split(",")
["apple", "peach", "banana", "pear"]
>>> s.split("na")
['apple,peach,ba', '', ',pear']

>>> s.split("ana")
['apple,peach,b', 'na,pear']
>>> s.rsplit("ana")
['apple,peach,ban', ',pear']

>>> s.partition(',')
('apple', ',', 'peach,banana,pear')
>>> s.rpartition(',')
('apple,peach,banana', ',', 'pear')
```

‘,’ is removed

Note empty string ''

left to right
vs
right to left

以左第一个分隔

‘,’ is preserved

String

- **String-specific methods (concatenate)** 连接
  - **__add__(self, other): return a new string by concatenating two strings**

$$s_1+s_2$$

  - **join(self, iterable): return a new string by concatenating any number of strings in which self is inserted in between each given string**

$$s.join([s_1,\ldots,s_n])$$

Join is more efficient than +

String-specific methods

```
import time
s = ['This is a long string that will not
keep in memory.' for n in range(10000)]

start = time.process_time()
x = " ".join(s)
print(time.process_time() - start)

start = time.process_time()
y = ""
for i in s:
    y=y + " " + i
print(time.process_time() - start)
```

Output

```
0.0
0.625
```

```
isalnum(self, /): return True if the string is an alpha-numeric string and non-empty, False otherwise

isalpha(self, /): return True if the string is an alphabetic string and non-empty, False otherwise.

isascii(self, /): return True if all characters in the string are ASCII or empty, False otherwise

isidentifier(self, /): return True if the string is a valid Python identifier such as "def" and "class", False otherwise

isdecimal(self, /): return True if the string is a decimal string and non-empty, False otherwise

isdigit(self, /): return True if the string is a digit string and non-empty, False otherwise

isnumeric(self, /): return True if the string is a numeric string and non-empty, False otherwise
```

/ means only positional arguments rather than keyword arguments

decimals ⊂ digits ⊂ numeric

String-specific methods

```
num = "0201"
print(num.isdigit())   # True
print(num.isdecimal()) # True
print(num.isnumeric()) # True

num = "①"
print(num.isdigit())   # True
print(num.isdecimal()) # False
print(num.isnumeric()) # True

num = "四"
print(num.isdigit())   # False
print(num.isdecimal()) # False
print(num.isnumeric()) # True

print("1.0".isnumeric()) # False
print("-1".isnumeric())  # False
```

小数

```
islower(self, /): return True if the string is a lowercase
string and non-empty, False otherwise.

isupper(self, /): return True if the string is an uppercase
string and non-empty, False otherwise.

istitle(self, /): return True if the string is a title-cased
string, False otherwise.

lower(self, /): return a copy of the string converted to
lowercase.

upper(self, /): return a copy of the string converted to
uppercase.

title(self, /): return a version of the string where each
word is titlecased.

swapcase(self, /): convert uppercase characters to lowercase
and lowercase characters to uppercase.
```

```
isspace(self, /): return True if the string is a whitespace
string and non-empty, False otherwise.

isprintable(self, /): return True if the string is
printable in repr() or empty, False otherwise.

startswith(self, prefix[, start[, end]]): return True if it
starts with the specified prefix, False otherwise.

ljust/rjust(self, width, fillchar=' ', /): return a
left/right-justified string of length width.

zfill(self, width, /): pad a numeric string with zeros on
the left, to fill a field of the given width.

lstrip/rstrip(self, chars=None, /): return a copy of the
string with leading/trailing whitespace removed. If chars
is given and not None, remove characters in chars instead.

strip(self, chars=None, /): return a copy of the string
with leading and trailing whitespace remove. If chars is
given and not None, remove characters in chars instead.
```

```
splitlines(self, /, keepends=False): return a list of the
lines in the string, breaking at line boundaries. '\n' are
not included in the resulting list unless keepends=True


replace(self, old, new, count=-1, /): return a copy with
first count occurrences of substring old replaced by new. -1
(default): replace all occurrences.


translate(self, table, /): replace each character in the
string using the given translation table, i.e., a mapping of
Unicode ordinals to Unicode ordinals, strings, or None.
Characters mapped to None are deleted.
```

Learning Objectives

- **Understand and use *compound* data types, used to group together other values**
  - **List**
  - **Tuple**
  - **String**
  - **Bytes and Bytearray**
  - **Set and Frozenset**
  - **Dictionaries**

Bytes and Bytearray

- **Bytes: immutable sequences of single ASCII bytes**
  - **Single quotes: b'still allows embedded "double" quotes'**
  - **Double quotes: b"still allows embedded 'single' quotes".**
  - **Triple quoted: b'''3 single quotes''', b"""3 double quotes"" "**

- **Bytearray: a mutable counterpart to bytes objects**

Bytearray(b”string”)

- **Bytearray supports the mutable sequence operations in addition to the common bytes and bytearray operations**

- **Bytes and Bytearray provides almost all methods of string, check via**

help(bytes) and help(bytearray)

Bytes vs Bytearray vs String

```
>>> x = "abc"
>>> y = b"abc"
>>> z = bytearray(b"abc")
>>> xd = x.replace(“a”,“d”)      # "dbc"
>>> yd = y.replace(b"a",b"d")     # b"dbc"
>>> zd = z.replace(b"a",b"d")     # bytarray("dbc“)
>>> x is xd                 # False
>>> y is yd                 # False
>>> z is zd                 # False
>>> z.extend(b"def")   # bytearray(b'abcdef')
>>> z.append(b"g") #TypeError:an int. is required
>>> z.append(103)   # bytearray(b'abcdefg')
```

Replace creates new object

Learning Objectives

- **Understand and use *compound* data types, used to group together other values**
  - **List**
  - **Tuple**
  - **String**
  - **Bytes**
  - **Bytearray**
  - **Set and Frozenset**
  - **Dictionaries**

Set and Frozenset

- **Set: an unordered mutable collection of distinct hashable and immutable objects**

  {1,2,'a','b'}

  Set() # empty set, not {}

- **Hashable: if it has a hash value which never changes during its lifetime (it needs a __hash__() method), and can be compared to other objects (it needs an __eq__() method)**
- **Common uses include**
  - membership testing
  - removing duplicates from a sequence
  - intersection, union, difference, and symmetric difference
- **Frozenset: an immutable counterpart to set, e.g, no add(), remove() methods, frozenset({1,2,3})**

Set and Frozenset

```
>>> a = {3, 'a', 2, 1,11,15,'1','2'}
>>> a.pop()
1
>>> a.remove(11)
>>> a
{2, 3, '2', '1', 15, 'a'}
>>> a.add(20)
>>> a
{2, 3, '2', '1', 15, 20, 'a'}
>>> b = {2,3}
>>> b.issubset(a)
True
>>> c = {5,6}
>>> c.isdisjoint(a)
True
>>> x = {1,1.0,1.00}
>>> x
{1}
```

pop: remove and return an arbitrary set element

pop and remove raise KeyError if the set does not contain the element

Elements are compared via ==

Set and Frozenset

```
>>> a_set = set([8, 9, 10, 11, 12, 13])
>>> b_set = {0, 1, 2, 3, 7, 8}
>>> a_set | b_set                    # union
{0, 1, 2, 3, 7, 8, 9, 10, 11, 12, 13}
>>> a_set.union(b_set)               # union
{0, 1, 2, 3, 7, 8, 9, 10, 11, 12, 13}
>>> a_set & b_set                  # intersection
{8}
>>> a_set.intersection(b_set)      # intersection
{8}
>>> a_set.difference(b_set)        # difference
{9, 10, 11, 12, 13}
>>> a_set - b_set
{9, 10, 11, 12, 13}
```

Learning Objectives

- **Understand and use *compound* data types, used to group together other values**
  - **List**
  - **Tuple**
  - **String**
  - **Bytes**
  - **Bytearray**
  - **Set and Frozenset**
  - **Dictionaries**

Dictionaries

- Dict: a mutable mapping object maps hashable values to arbitrary objects

  {Key:value pairs}

- Keys: hashable, unique (compare via ==)
- Values: can be not hashable
- Values are accessed via keys
  - d[key] : return value for key if exists, otherwise KeyError
  - d.get(key): return value for key if exists, otherwise None
  - d[key] = value: overwrite the value if key exists, otherwise add the new pair key:value
- d.items(): return key:value pairs
- d.keys(): return all the keys
- d.values(): return all the values

Dictionaries

```
>>> aDict={'name':'Fu', 'sex':'male', 'age':36}
>>> for item in aDict.items(): # print all pairs
print(item)

('age', 37)
('name', 'Fu')
('sex', 'male')

>>> for key in aDict: # in default, print keys
print(key)

age
name
sex
```

与 for 后面变量名无关

Dictionaries

```
>>> aDict={'name':'Fu', 'sex':'male', 'age':36}
>>> for key, value in aDict.items(): # unzip pairs
     print(key, value)

age 36
name Fu
sex male

>>> aDict.keys()   # return all keys
dict_keys(['name', 'sex', 'age'])

>>> aDict.values() # return all values
dict_values(['Fu', 'male', 36])
```

```
>>> aDict={'name':'Fu', 'sex':'male', 'age':36}
>>> aDict['school']='SIST'
>>> aDict
{'name': 'Fu', 'sex': 'male', 'age': 36, 'school':
'SIST'}
>>> del aDict['sex']
>>> aDict
{'name': 'Fu', 'age': 36, 'school': 'SIST'}
>>> aDict.update({'Courses':['CS100','CS131']})
>>> aDict
{'name': 'Fu', 'age': 36, 'school': 'SIST', 'Courses':
['CS100', 'CS131']}
>>> aDict.pop("Courses")
['CS100', 'CS131']
>>> aDict
{'name': 'Fu', 'age': 36, 'school': 'SIST'}
>>> aDict.clear()
>>> aDict
{}
```

Recap

- **Understand and use *compound* data types, used to group together other values**
  - **List**
  - **Tuple**
  - **String**
  - **Bytes**
  - **Bytearray**
  - **Set and Frozenset**
  - **Dictionaries**