Python for Data Analytics

Python
Sequence and Collections

Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Jan. 3 – 14, 2022

Sequence vs. Collection

- Both can hold a bunch of values in a single "variable"

- Sequence
  - Sequence has a deterministic ordering
  - Index their entries based on the position
  - Strings, Lists, Tuples

- Collection
  - No ordering
  - Sets, Dictionaries

Lists

Lists

- Ordered collections of arbitrary objects (arrays of object references)
- Accessed by offset (items not sorted)
- Variable-length, heterogeneous, mutable, and arbitrarily nestable
- The most versatile and popular data type in Python

```
prime = [2, 3, 5, 7, 11]
a = [2, 'three', 3.0, 5, 'seven', 11.0]
b = [1, 3, 3, 3, 2, 2]
c = [ 1, [8, 9], 12]
emptylist = []
```

Concatenating/Replicating Lists

- *list1* + *list2* : create a new list by adding two existing lists together
- *list* * *n*: create a new list by replicating the original list *n* times

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
>>> d = a*3
>>> print(d)
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Referencing a List Element

- Just like strings, use an index specified in square brackets

Emma	Olivia	Ava	Isabella	Sophia
0	1	2	3	4

```
>>> names = ['Emma', 'Olivia', 'Ava', 'Isabella', 'Sophia']
>>> print(names[0])
Emma
>>> print(names[2])
Ava
```

Slicing a List

- *list*[*start* : *end* : *step*]
  - *start*: the starting index of the list
    - If omitted, the beginning of the list if *step* > 0 or the end of the list if step < 0
  - *end*: the ending index of the list (up to but not including)
    - If omitted, the end of the list if *step* > 0 or the beginning of the list if step < 0
  - *step*: the number of elements to skip + 1 (1 if omitted)
  - *start* and *stop* can be a negative number, which means it counts from the end of the array

Emma	Olivia	Ava	Isabella	Sophia
0	1	2	3	4
-5	-4	-3	-2	-1

```
names = ['Emma', 'Olivia', 'Ava', 'Isabella', 'Sophia']
```

Slicing a List: Example

Emma	Olivia	Ava	Isabella	Sophia
0	1	2	3	4
-5	-4	-3	-2	-1

```
>>> names = ['Emma', 'Olivia', 'Ava', 'Isabella', 'Sophia']
>>> print(names[::2])
['Emma', 'Ava', 'Sophia']
>>> print(names[3:])
['Isabella', 'Sophia']
>>> print(names[-3:])
['Ava', 'Isabella', 'Sophia']
>> print(names[::-1])
['Sophia', 'Isabella', 'Ava', 'Olivia', 'Emma']
```

Slicing a List: Example (cont'd)

Emma	Olivia	Ava	Isabella	Sophia
0	1	2	3	4
-5	-4	-3	-2	-1

```
>>> names = ['Emma', 'Olivia', 'Ava', 'Isabella', 'Sophia']
>>> print(names[])                         # WRONG!

>>> print(names[:])


>>> print(names[::])


>>> print(names[-10:10])
```

Getting the List Size

- `len(`*list*`)`
  - Return the number of elements in the list
  - Actually, `len()` tells us the number of elements of any sequence or collection (e.g., string, list, tuple, dict, set, …)

```
>>> greet = 'Hello Spam'
>>> print(len(greet))
10
>>> x = [ 1, 2, 'spam', 99, 'ham' ]
>>> print(len(x))
5
```

Lists are Mutable

- *list*[*i*] = *x*: change the *i*-th element of the list to *x*
- *list*[*i*:*j*] = *list2*: replace the elements from *i*-th to *j-th* with the new list

```
>>> names = ['Emma', 'Olivia', 'Ava', 'Isabella', 'Sophia']
>>> names[2] = 'Eve'
>>> print(names)
['Emma', 'Olivia', 'Eve', 'Isabella', 'Sophia']
>>> names[1:4] = [ 'Charlotte', 'Mia', 'Amelia']
>>> print(names)
['Emma', 'Charlotte', 'Mia', 'Amelia', 'Sophia']
>>> names[5:] = [ 'Ella', 'Avery' ]
>>> print(names)
['Emma', 'Charlotte', 'Mia', 'Amelia', 'Sophia', 'Ella', 'Avery']
```

Useful Functions on a List

- *list*.count(*x*)
  - Return the number of times *x* appears in the list
- max(*list*)
  - Return the maximum value in the list
- min(*list*)
  - Return the minimum value in the list

```
>>> numbers = [3, 1, 12, 14, 12, 6, 1, 12]
>>> print(numbers.count(12))
3
>>> print(min(numbers), max(numbers))
1 14
```

Finding an Element

- *list*.index(*x*[, *start*[, *end*]])
  - Return zero-based index in the list of the first item whose value is equal to *x*
  - IndexError if there is no such item
  - The optional *start* and *end* arguments are used to limit the search to a particular subsequence of the list

```
>>> names = ['Emma', 'Olivia', 'Ava', 'Emma', 'Isabella']
>>> print(names.index('Emma')
0
>>> print(names.index('Emma', 1, 4))
3
>>> print(names.index('Isabella', 3))
4
```

Building a List

- *list*.append(*x*)
  - Add an item *x* to the existing list
  - The list stays in order and new elements are appended at the end of the list

```
>>> menu = list()
>>> print(menu)
[]
>>> menu.append('spam')
>>> menu.append('ham')
>>> menu.append('spam')
>>> print(menu)
['spam', 'ham', 'spam']
```

Extending Lists

- *list*.extend(*list2*)
  - Extend the list by appending all the items from the other list
  - Faster than a series of append()'s

```
>>> menu = ['spam', 'ham']
>>> menu.extend(['egg', 'sausage'])
>>> print(menu)
['spam', 'ham', 'egg', 'sausage']
>>> menu.extend(['spam']*3)
>>> print(menu)
['spam', 'ham', 'egg', 'sausage', 'spam', 'spam', 'spam']
```

Inserting an Element

- *list*.`insert(`*i*, *x*`)`
  - Insert an item at a given position *i* (0 means the front of the list)

```
>>> menu = ['spam', 'ham']
>>> print(menu)
['spam', 'ham']
>>> menu.insert(1, 'egg')
>>> print(menu)
['spam', 'egg', 'ham']
>>> menu.insert(0, 'bacon')
>>> print(menu)
['bacon', 'spam', 'egg', 'ham']
```

Removing Elements

- *list*.remove(*x*)
  - Remove the first item from the list whose value is equal to *x*
- del *list*[*i*] or del *list*[*i*:*j*]
  - Deletes *i*-th element (or from *i*-th to *j*-1th elements) from the list

```
>>> menu = ['spam', 'ham', 'egg', 'sausage', 'bacon']
>>> menu.remove('ham')
>>> print(menu)
['spam', 'egg', 'sausage', 'bacon']
>>> del menu[1:3]
>>> print(menu)
['spam', 'bacon']
```

Popping an Element

- *list*.pop([*i*])
  - Remove the item at the given *i*-th position in the list, and return it
  - If no index is specified, it removes and returns the last item in the list

```
>>> menu = ['spam', 'ham', 'egg', 'sausage', 'bacon']
>>> print(menu.pop(1))
ham
>>> print(menu.pop())
bacon
>>> print(menu.pop())
sausage
>>> print(menu.pop())
egg
```

Reversing the List

- *list*.`reverse()`
  - Reverse the elements of the list in place
  - cf. *list*[::-1] returns the new list with the elements in reversed order

```
>>> names = ['Emma', 'Olivia', 'Ava', 'Isabella', 'Sophia']
>>> names.reverse()
>>> print(names)
['Sophia', 'Isabella', 'Ava', 'Olivia', 'Emma']
>>> new_names = names[::-1]
>>> print(new_names)
['Emma', 'Olivia', 'Ava', 'Isabella', 'Sophia']
```

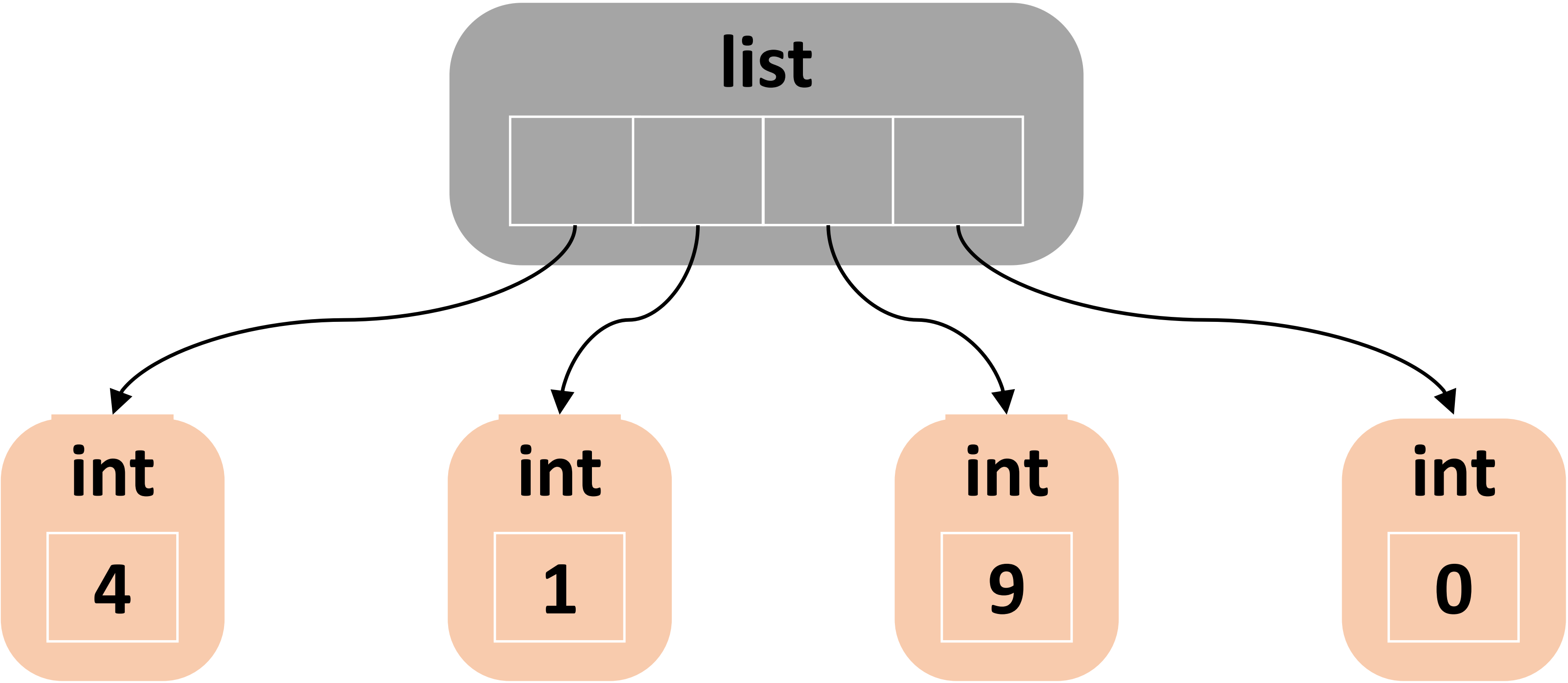
Membership Operators

- in (not in) operator
  - Check if an item is in a list or not
  - Returns True or False

```
>>> menu = ['spam', 'ham']
>>> 'spam' in menu
True
>>> 'ham' not in menu
False
>>> 'egg' in menu
False
```

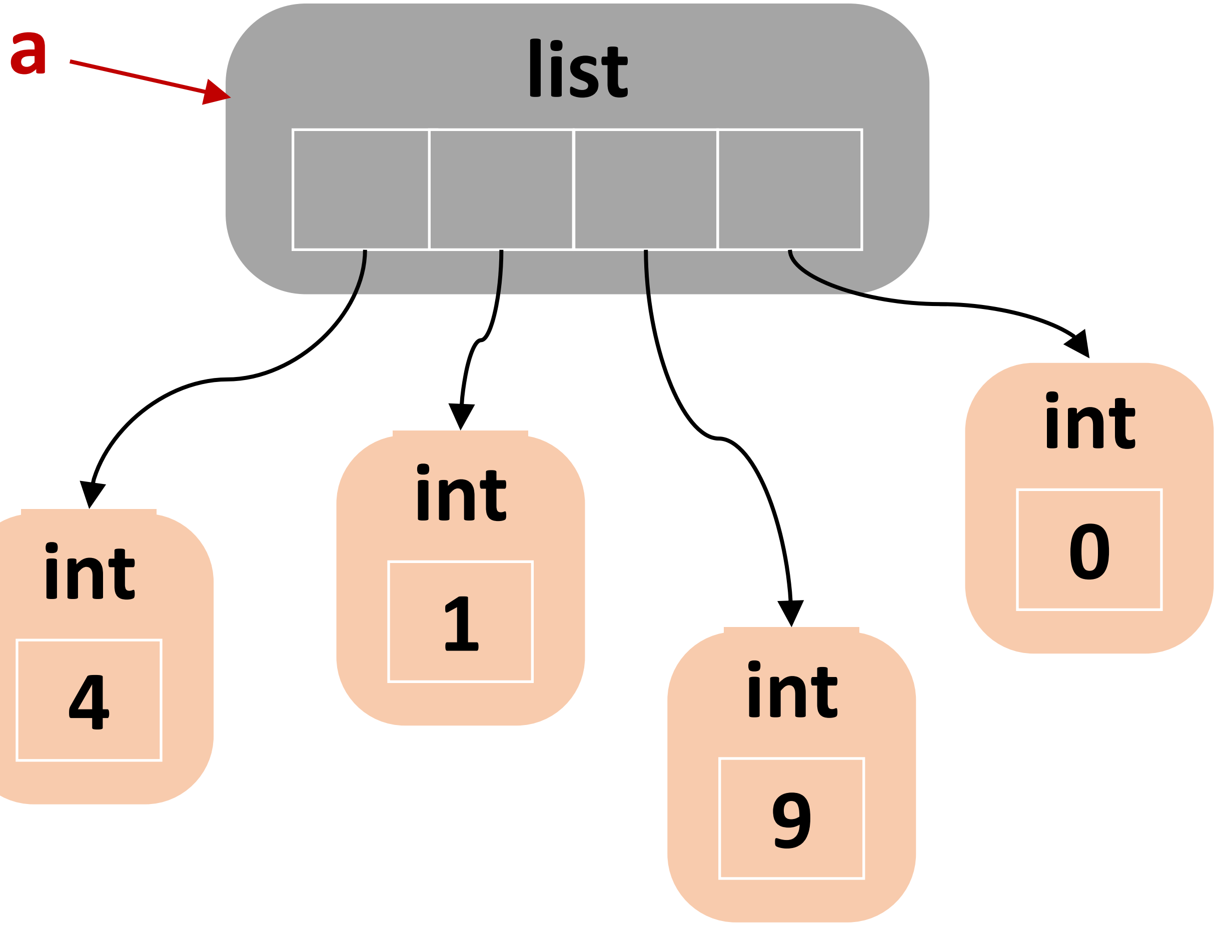
Implementing Lists

- A list has an array of references to other objects



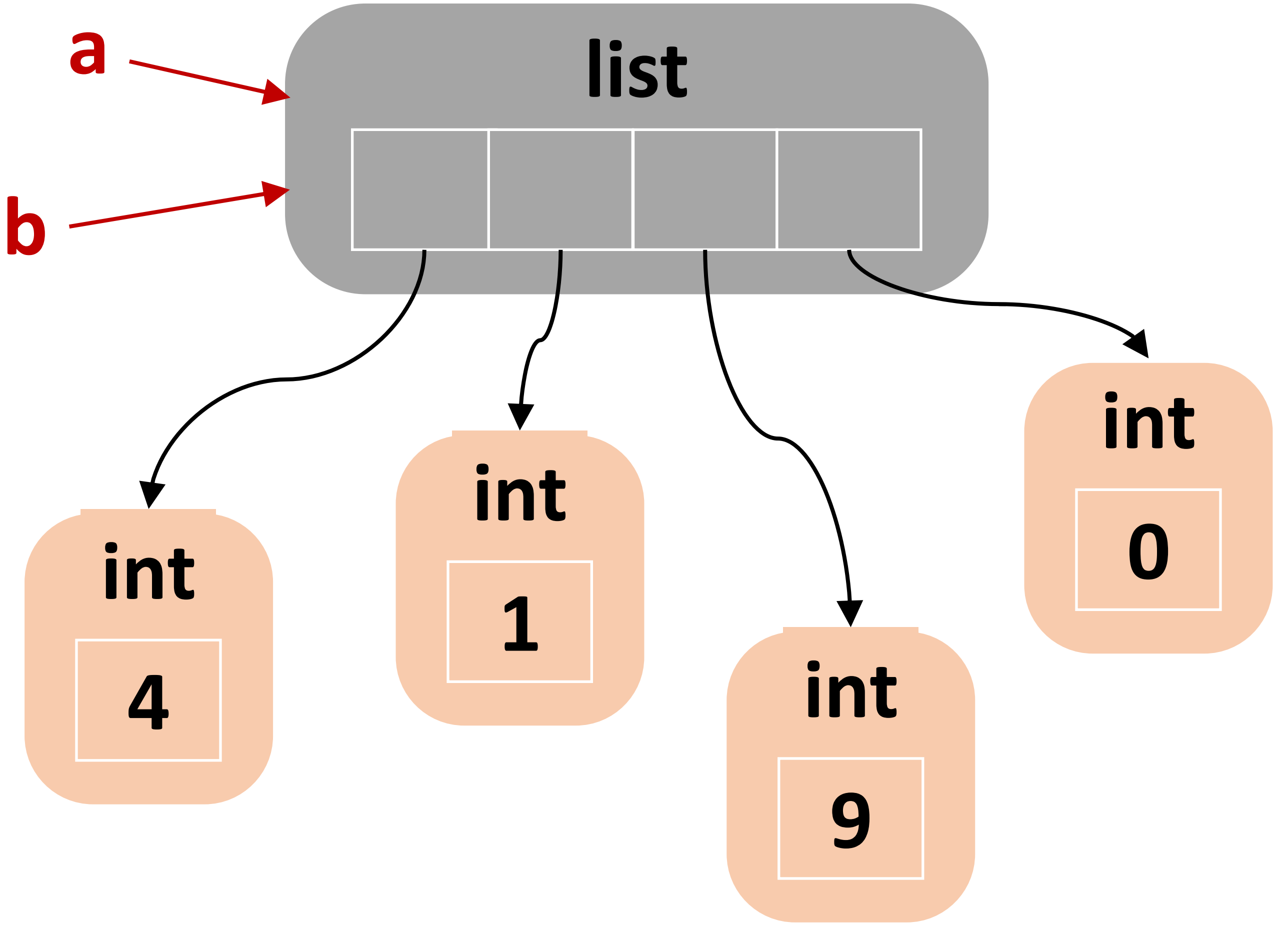
Assigning a List

```
>>> a = [4, 1, 9, 0]
```



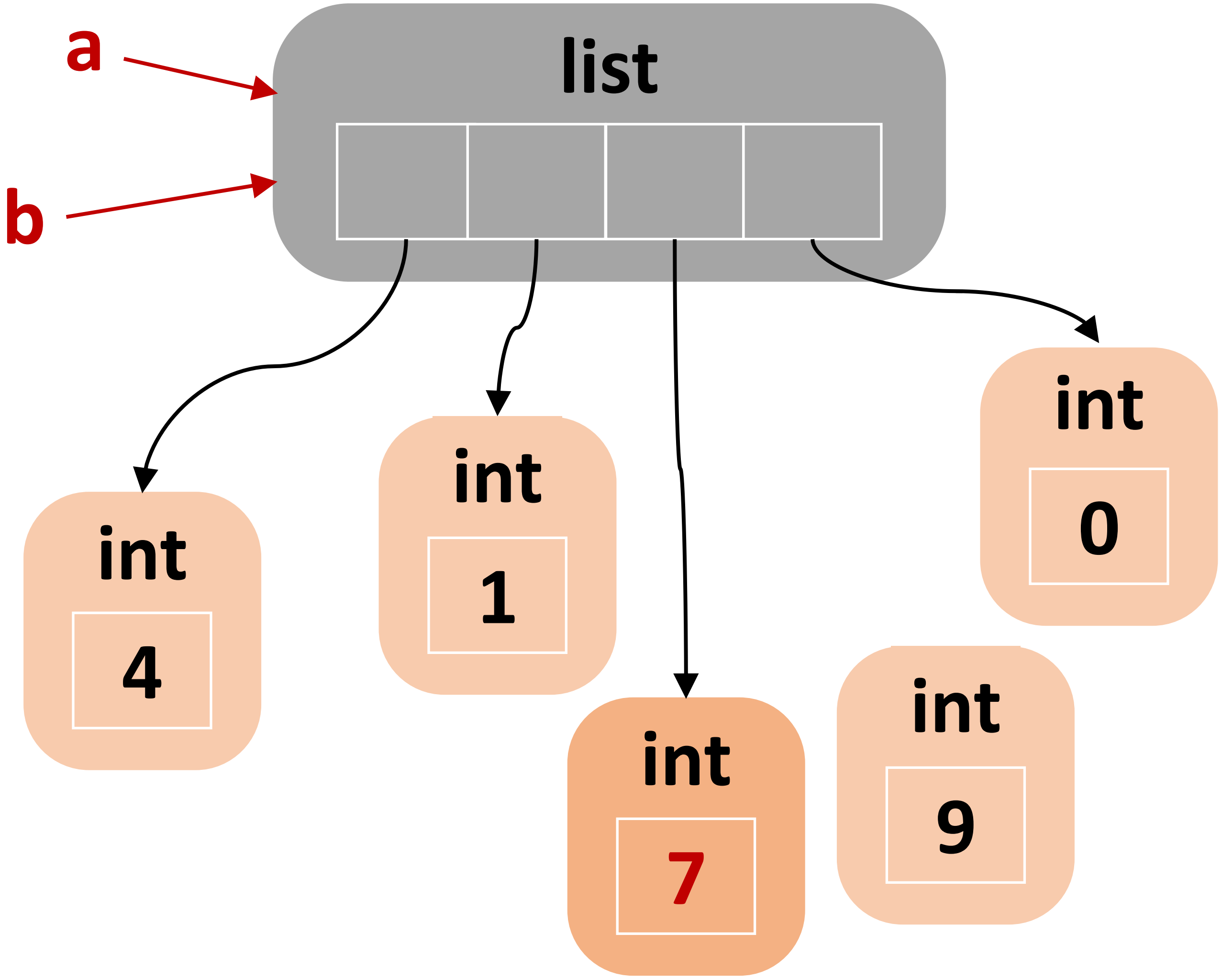
Assigning a List

```
>>> a = [4, 1, 9, 0]
>>> b = a
```



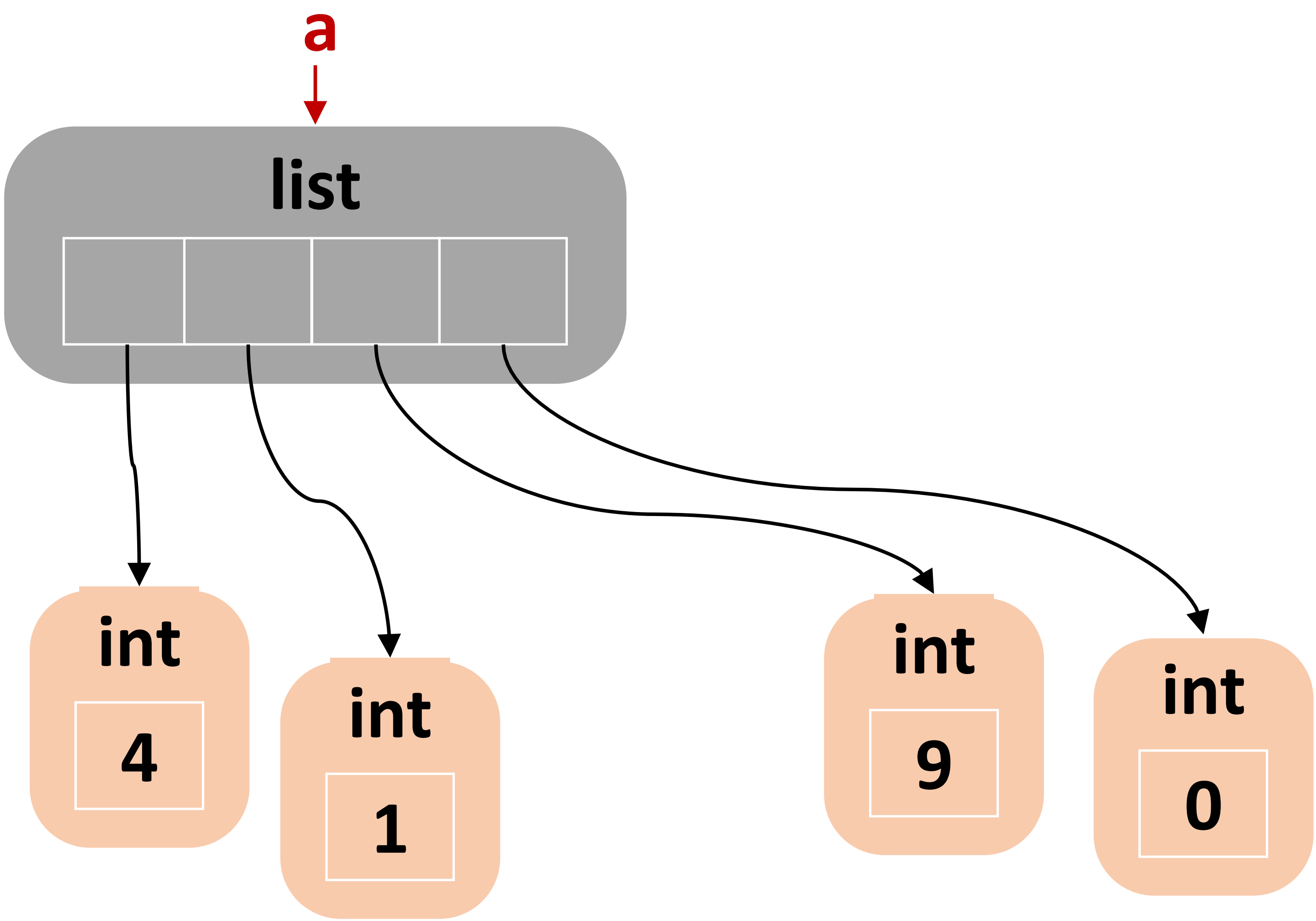
Changing a List Element

```
>>> a = [4, 1, 9, 0]
>>> b = a
>>> a[2] = 7
>>> print(a)
[4, 1, 7, 0]
>>> print(b)
[4, 1, 7, 0]
```



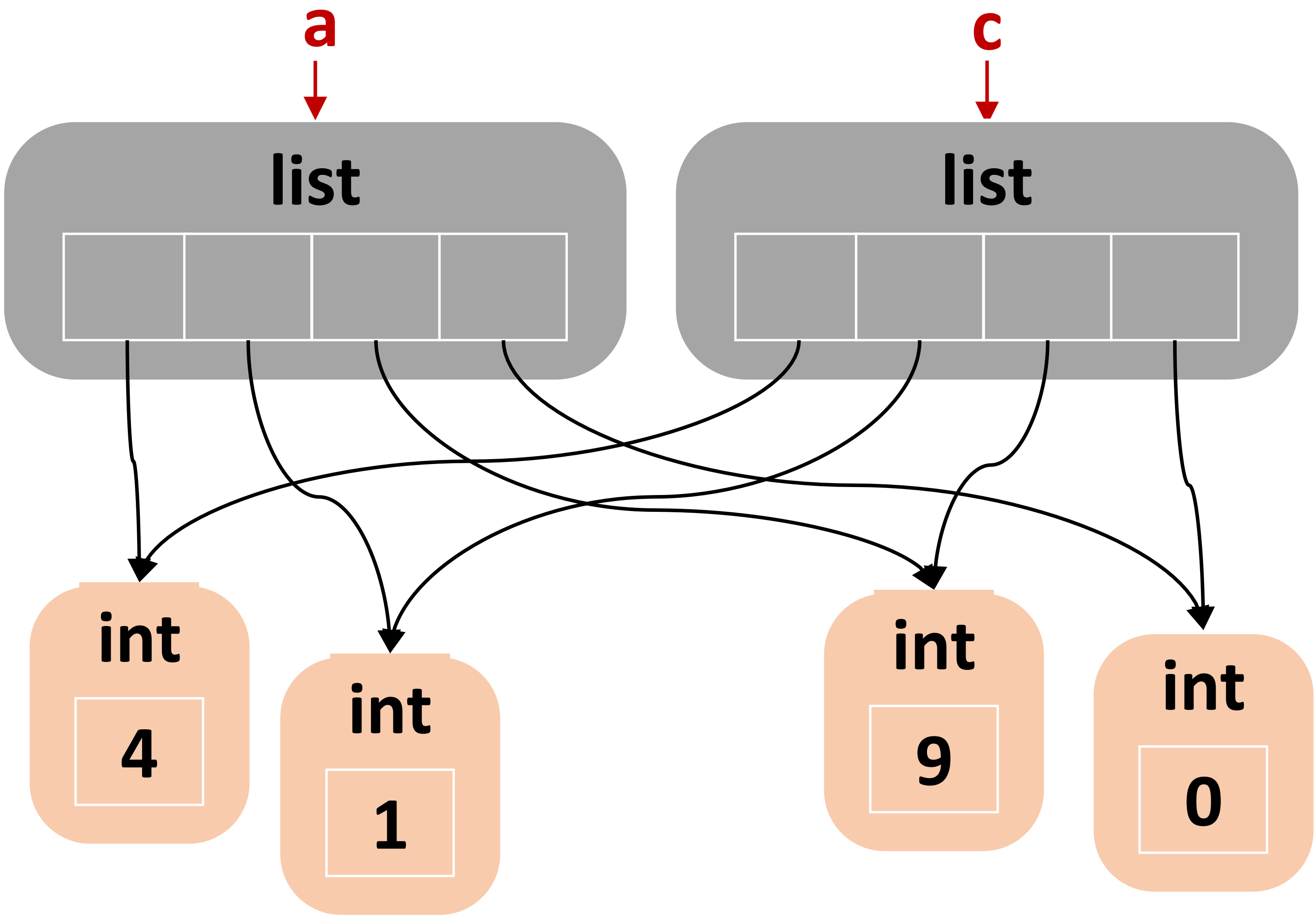
Copying a List

```
>>> a = [4, 1, 9, 0]
```



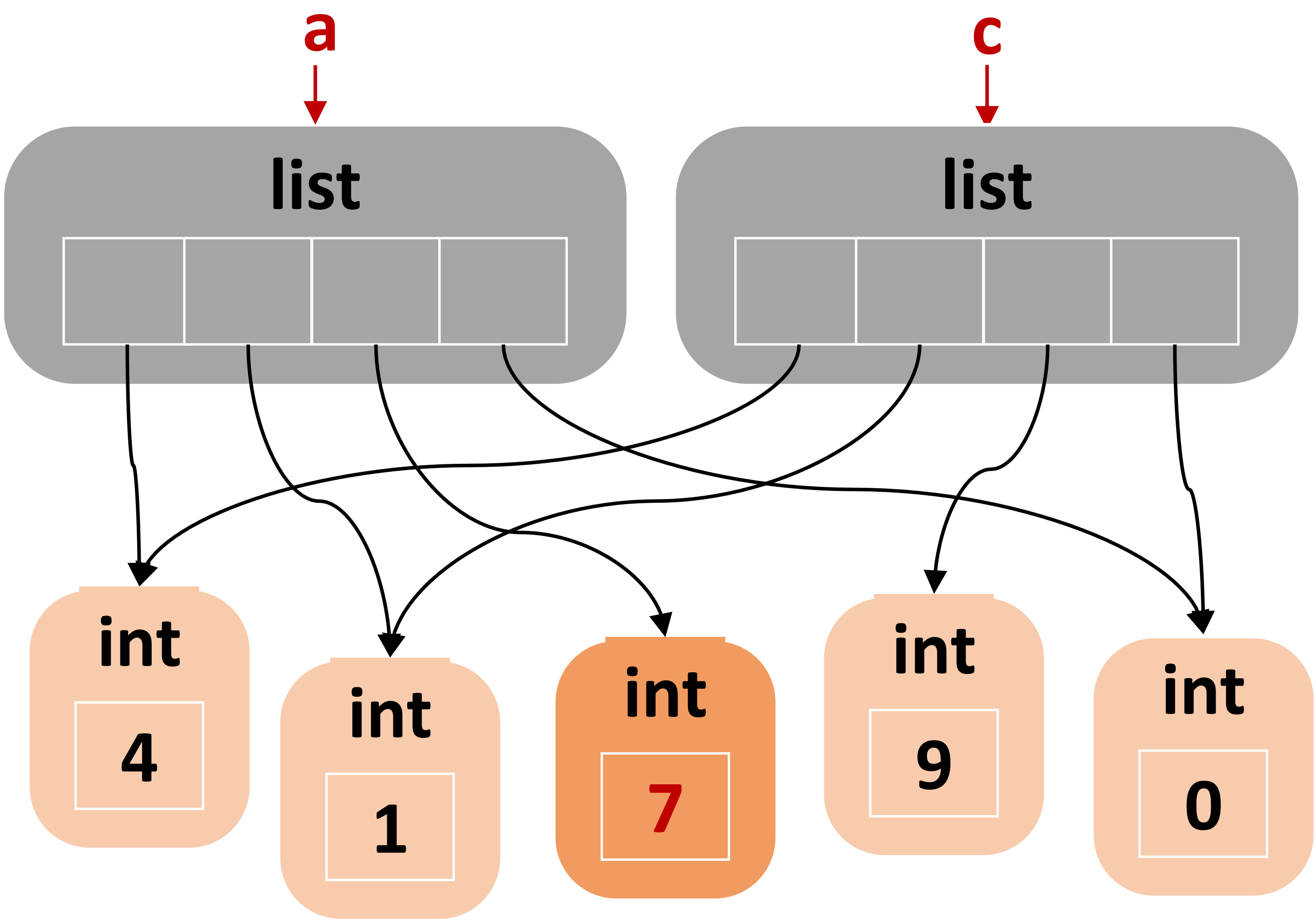
Copying a List

```
>>> a = [4, 1, 9, 0]
>>> c = a[:]
```



Copying a List

```
>>> a = [4, 1, 9, 0]
>>> c = a[:]
>>> a[2] = 7
>>> print(a)
[4, 1, 7, 0]
>>> print(c)
[4, 1, 9, 0]
```



Copying a List: Other Ways

- Using list slicing

```
>>> a = [1, 2, 3, 4]
>>> b = a[:]
```

- Using *

```
>>> a = [1, 2, 3, 4]
>>> b = a*1
```

- Using `list()`

```
>>> a = [1, 2, 3, 4]
>>> b = list(a)
```

- Using copy module

```
>>> import copy
>>> a = [1, 2, 3, 4]
>>> b = copy.copy(a)
```

Sorting Elements in a List (1)

- *list*.sort([*key*], [*reverse*])
  - Sort the elements in the list
  - *key*: a function with a single argument (used to extract a comparison key)
  - *reverse*: if True, the list elements are sorted in the reverse order
  - *list*.sort() changes the list in place, but don't return the list as a result

```
>>> names = ['Emma', 'Olivia', 'Ava', 'Isabella', 'Sophia']
>>> names.sort()
>>> print(names)
['Ava', 'Emma', 'Isabella', 'Olivia', 'Sophia']
>>> names.sort(reverse=True)
['Sophia', 'Olivia', 'Isabella', 'Emma', 'Ava']
```

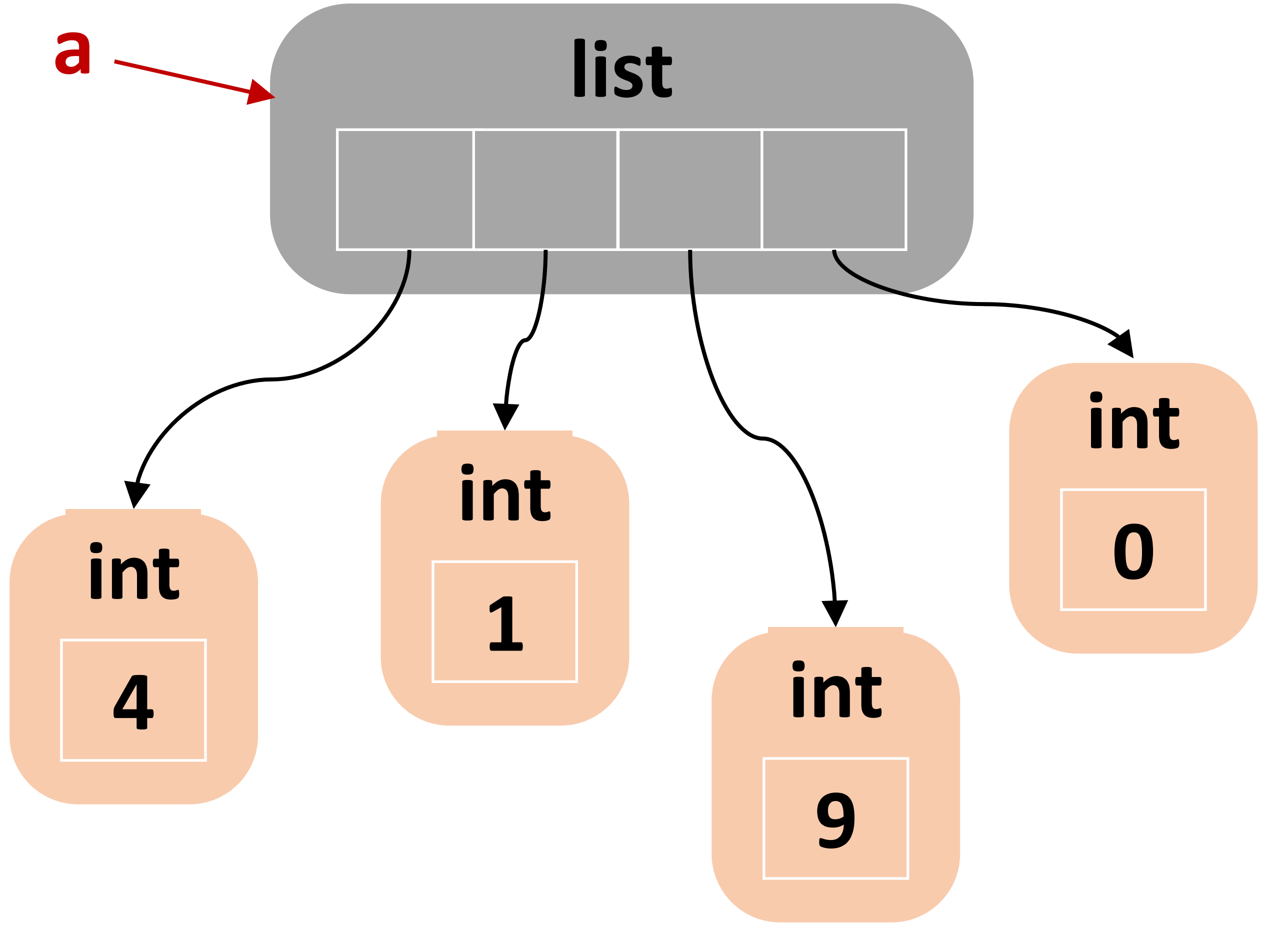
Sorting Elements in a List (2)

- sorted(*iterable*, [*key*], [*reverse*])
  - Sort the elements in the list
  - *key*: a function with a single argument (used to extract a comparison key from each element)
  - *reverse*: if True, the list elements are sorted in the reverse order
  - sorted() returns a new list!

```
>>> names = ['Emma', 'Olivia', 'Ava', 'Isabella', 'Sophia']
>>> sorted_names = sorted(names)
>>> print(sorted_names)
['Ava', 'Emma', 'Isabella', 'Olivia', 'Sophia']
```

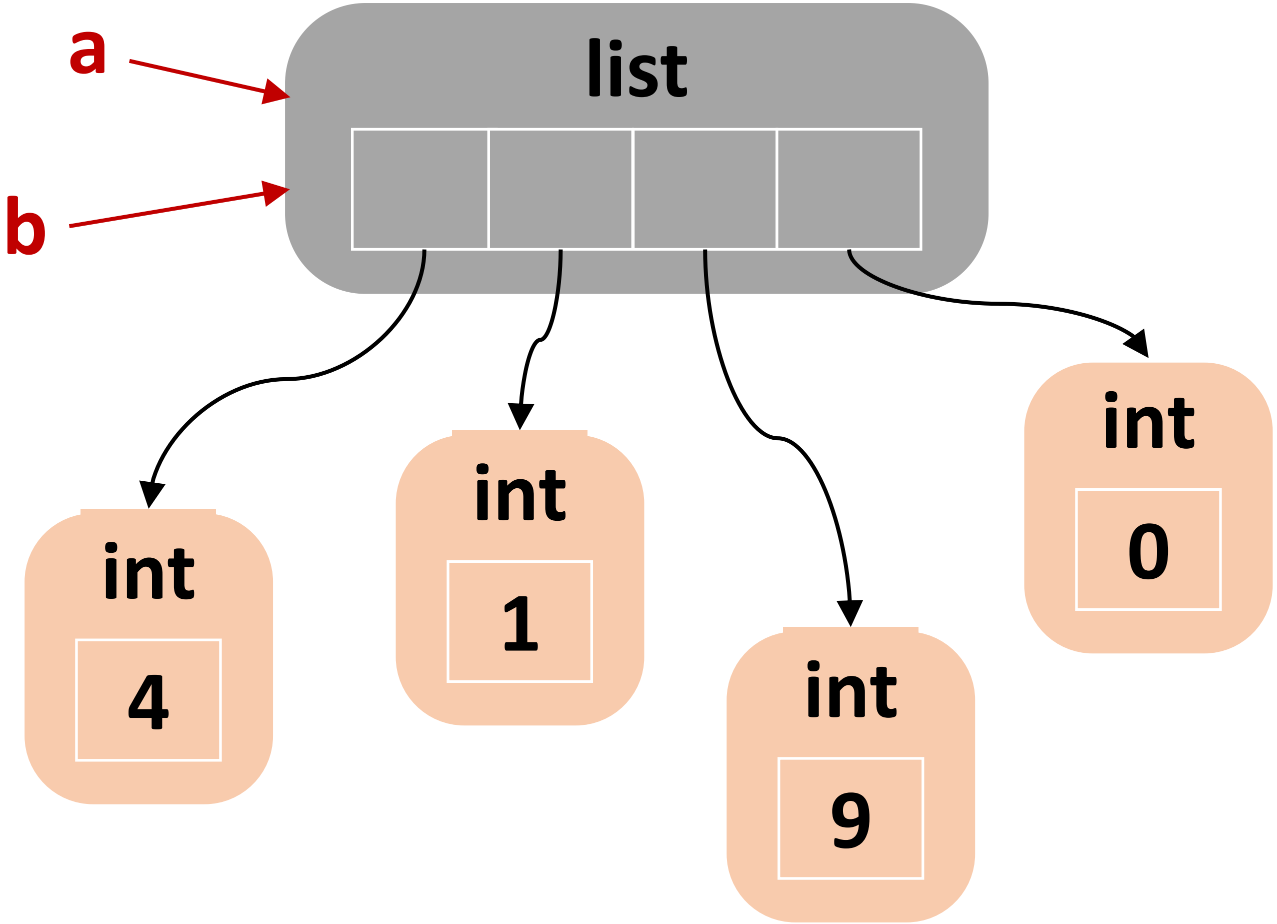
list.sort()

```
>>> a = [4, 1, 9, 0]
```



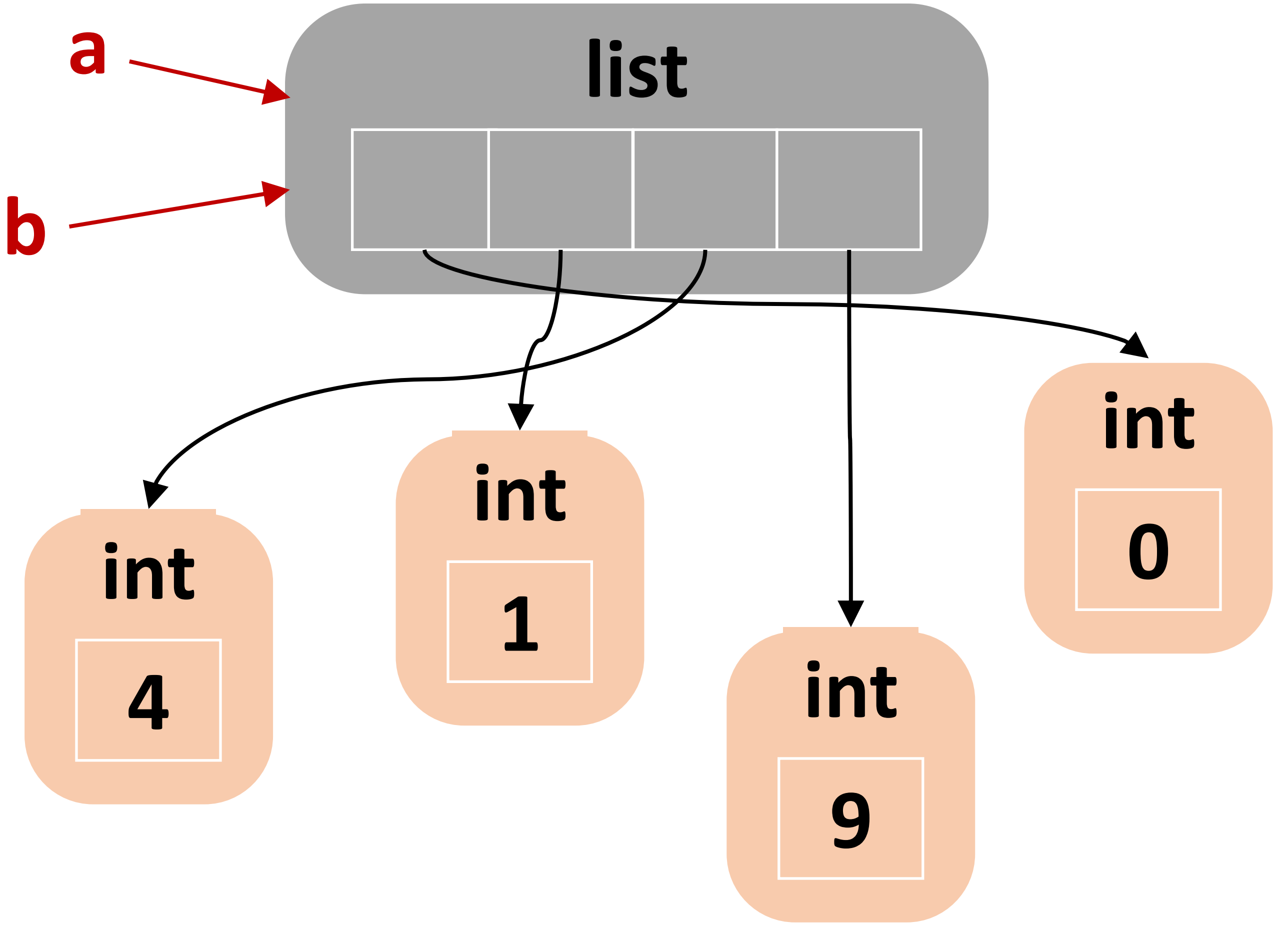
list.sort()

```
>>> a = [4, 1, 9, 0]
>>> b = a
```



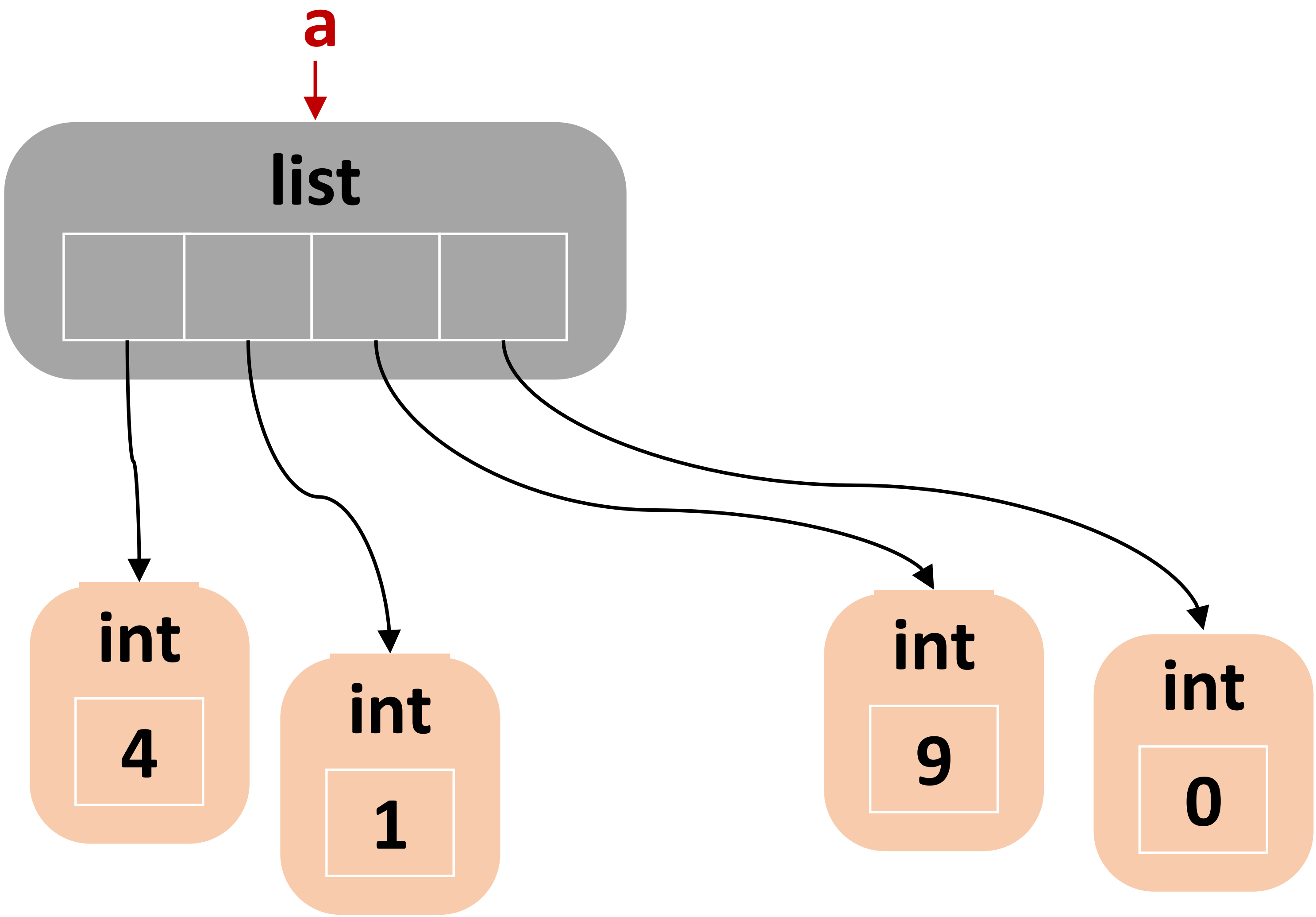
list.sort()

```
>>> a = [4, 1, 9, 0]
>>> b = a
>>> b.sort()
>>> print(b)
[0, 1, 4, 9]
>>> print(a)
[0, 1, 4, 9]
```



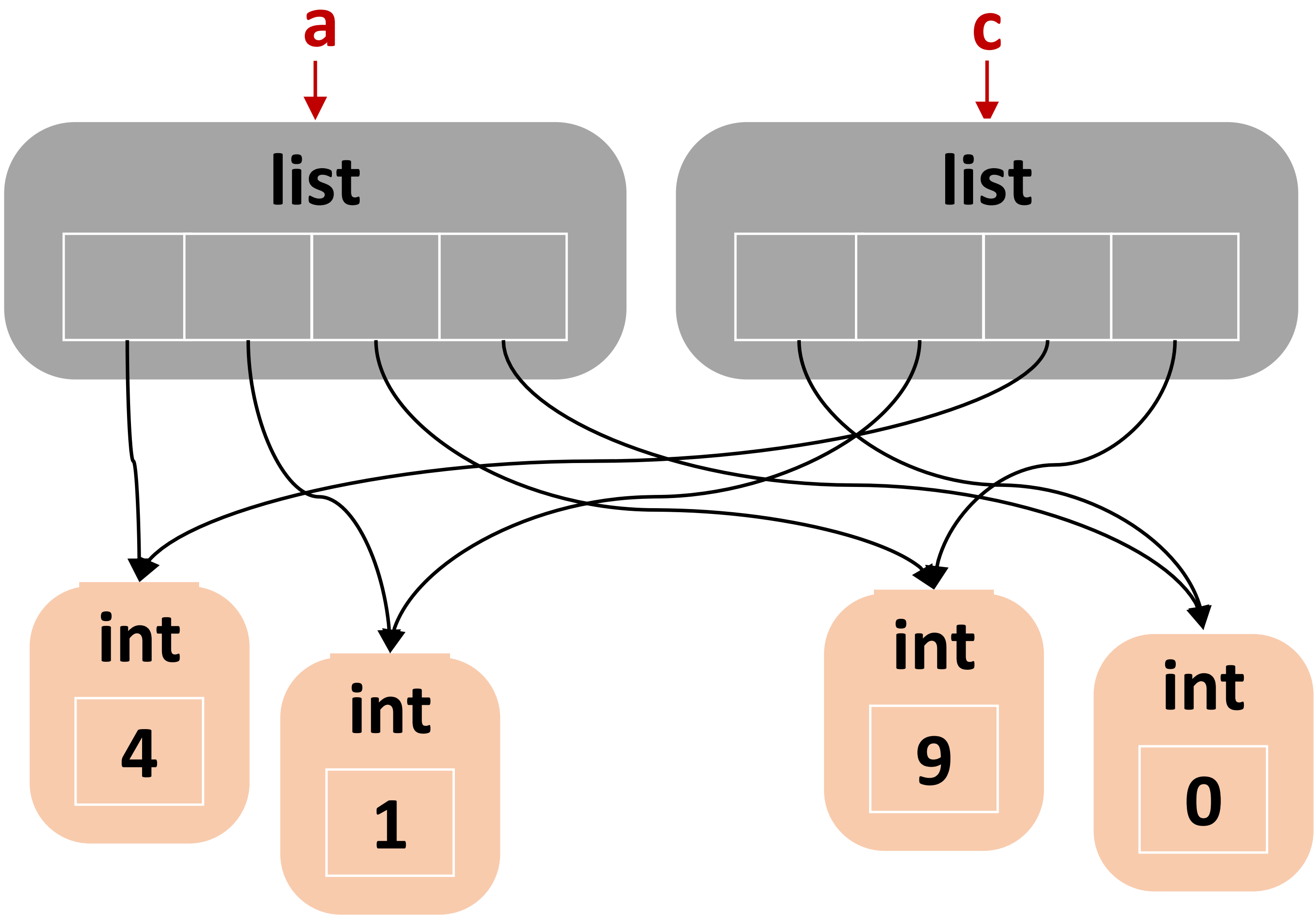
sorted(list)

```
>>> a = [4, 1, 9, 0]
```



sorted(list)

```
>>> a = [4, 1, 9, 0]
>>> c = sorted(a)
>>> print(c)
[0, 1, 4, 9]
>>> print(a)
[4, 1, 9, 0]
```



Multi-dimensional Lists

- Lists within lists

```
>>> M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M[1]
[4, 5, 6]
>>> M[1][2]
6
>>> M[1:]
[[4, 5, 6], [7, 8, 9]]
>>> M[2] = [0, 0, 0]
>>> M
[[1, 2, 3], [4, 5, 6], [0, 0, 0]]
```

M x N Matrix using Lists

- M x N Matrix: M rows and N columns

```
>>> M = [[1, 2, 3], [4, 5, 6]]     # 2 x 3 matrix
>>> len(M)                         # number of rows
2
>>> len(M[0])                      # number of columns
3
>>> M[1][0]                        # element M(1,0)
4
>>> M[0][0]+M[0][1]+M[0][2]        # sum of first row
6
>>> M[0][1]+M[1][1]                # sum of second column
7
```

Accessing Multi-dimensional Lists

```
M = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]

for row in M:
    print(row)

for i in range(len(M)):
    for j in range(len(M[i])):
        print(M[i][j], end=' ')

for row in M:
    for col in row:
        print(col, end=' ')
```

List Comprehension

- Provides a concise way to create lists

```
new_list = list()
for i in range(10):
    if i % 2 == 0:
        new_list.append(i*i)
```

```
new_list = [ i*i for i in range(10) if i % 2 == 0 ]
```

List Comprehension: General Form

```
new_list = [ expression(i) for i in sequence if filter(i) ]
```

```
new_list = list()
for i in sequence:
    if filter(i):
        new_list.append(expression(i))
```

Simple Lists

```
>>> [0] * 10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> [ i for i in range(10, 20) ]                  # list(range(10, 20))
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> [ x**2 for x in range(10) ]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [ i for i in range(100) if i % 3 == 0 and i % 5 == 1 ]
[6, 21, 36, 51, 66, 81, 96]
>>> [ random.randint(0, 99) for _ in range(10) ]   # import random
[50, 15, 22, 3, 88, 50, 71, 63, 40, 62]
```

Lists From Lists

```
>>> [ item*3 for item in [2, 3, 5] ]
[6, 9 ,15]
>>> [ i if i > 0 else 0 for i in [-2, 5, 4, -7] ]
[0, 5, 4, 0]
>>> [ word[0] for word in ['hello', 'world', 'spam'] ]
['h', 'w', 's']
>>> [ x.upper() for x in ['spam', 'ham', 'egg'] ]
['SPAM', 'HAM', 'EGG']
>>> [ x + y for x in [10, 30, 50] for y in [20, 40, 60] ]
[30, 50, 70, 50, 70, 90, 70, 90, 110]
```

Nested Lists

```
>>> [ [0]*4 for _ in range(3) ]                 # [[0]*4]*3 ??
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> [ [i for i in range(4)] for _ in range(3) ]
[[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]
>>> [ [x, y] for x in [1, 2, 3] for y in [7, 8, 9] ]
[[1, 7], [1, 8], [1, 9], [2, 7], [2, 8], [2, 9], [3, 7], [3, 8], [3, 9]]
>>> [ [[x, y] for x in [1, 2, 3]] for y in [7, 8] ]
[[[1, 7], [2, 7], [3, 7]], [[1, 8], [2, 8], [3, 8]]]
```

Nested Lists (2)

```
>>> matrix = [ [i for i in range(j, j+4)] for j in range(0, 12, 4)]
>>> matrix
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
>>> [ e for row in matrix for e in row ]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> for row in matrix:
...     print('\t'.join([str(e) for e in row]))
0       1       2       3
4       5       6       7
8       9       10      11
>>> print('\n'.join(['\t'.join([str(e) for e in row]) for row in matrix]))
```

zip()

- zip(*iterables)
  - Make an iterator that aggregates elements from each of the *iterables*
  - The * operator can be used to unzip a list

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> xy = list(zip(x,y))
>>> xy
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*xy)
>>> x2
(1, 2, 3)
>>> list(y2)
[4, 5, 6]
```

map()

- map(*func*, **iterables*)
  - Returns a map object (which is an iterator) of the results after applying the given function *func* to each item of a given *iterables*

```
>>> def double(x):
...     return x + x
>>> n = [1, 2, 3, 4]
>>> print(list(map(double, n)))
[2, 4, 6, 8]
>>> print(list(map(lambda x: x + x, n)
[2, 4, 6, 8]
>>> n2 = [10, 20, 30, 40]
>>> print(list(map(lambda x, y: x + y, n, n2)
[11, 22, 33, 44]
```

Tuples

Tuples

- Ordered collection of arbitrary objects
- Accessed by offset
- Immutable sequence
- Fixed-length, heterogeneous, and arbitrarily nestable

```
menu = (1, 2, 5, 9)
a = 1, 2, 5, 9
b = (0, 'ham', 3.14, 99)
c = ('a', ('x', 'y'), 'z')
emptytuple = ()              # tuple()
```

Tuples are like Lists

- Another kind of "sequence"
- Elements are indexed starting at 0

```
>>> num = (4, 1, 9)
>>> print(num[2])
9
>>> print(len(num))
4
>>> print(max(num))
9
>>> print(min(num))
1
```

```
>>> for i in num:
...     print(i)
4
1
9
>>> print(num + ('a', 'b'))
(4, 1, 9, 'a', 'b')
>>> print(num * 2)
(4, 1, 9, 4, 1, 9)
```

Tuples are Immutable

- Unlike a list, once you create a tuple, you cannot alter its contents
- Similar to a string

Lists

```
>>> x = [9, 8, 7]
>>> x[2] = 6
>>> print(x)
[9, 8, 6]
```

Strings

```
>>> s = 'ABC'
>>> s[2] = 'D'
Traceback (most recent
call last):
  File "<stdin>", line 1,
in <module>
TypeError: 'str' object
does not support item
assignment
```

Tuples

```
>>> z = (5, 4, 3)
>>> z[2] = 0
Traceback (most recent
call last):
  File "<stdin>", line 1,
in <module>
TypeError: 'tuple' object
does not support item
assignment
```

Things not to do with Tuples

```
>>> x = (4, 1, 9, 0)
>>> x.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'sort'
>>> x.append(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> x.reverse()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'reverse'
```

Tuples and Assignments

- We can also put a tuple on the left-hand side of an assignment statement
- We can even omit the parentheses
- Can be used to return multiple values in a function

```
>>> (x, y) = (4, 'spam')
>>> print(x)
4
>>> y = 1
>>> x, y = y, x
>>> print(x, y)
1 4
```

```
def ret2(a):
    return min(a), max(a)

a, b = ret2([4, 1, 9, 0])
```

Tuples are Comparable

- The comparison operators work with tuples and other sequences
- If the first item is equal, Python goes on to the next element, and so on, until it finds elements that differ

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> ('Jones', 'Sally') < ('Jones', 'Sam')
True
>>> ('Jones', 'Sally') > ('Adams', 'Sam')
True
```

Sets

Sets

- Unordered collection of unique immutable objects
- Not ordered (cannot be accessed by offset)
- Items can be added or removed
- Variable-length and heterogeneous, but not nestable
- Support operations corresponding to mathematical set theory

```
choice = {1, 2, 5, 9}
s = {'a', 'b', 'c', 'c'}  # ???
t = {0, 'ham', 3.14, 99}
emptyset = set()          # wrong: s = {}
```

Set Manipulation Operations

- `s.pop()` remove and return an arbitrary element from s
- `s.clear()` remove all elements from set s
- `s.add(x)` add element x to set s
- `s.remove(x)` remove x from set s
  raise KeyError if not present
- `s.discard(x)` remove x from set s if present

Mathematical Set Operations

`s.issubset(t)`	True if s ⊂ t (or `s <= t`)
`s.issuperset(t)`	True if s ⊃ t (or `s >= t`)
`s.union(t)`	return s ∪ t (or `s | t`)
`s.intersection(t)`	return s ∩ t (or `s & t`)
`s.difference(t)`	return `s - t`
`s.symmetric_difference(t)`	return `t - s`

Set Membership Check is Fast!

```python
import time

N = 10000
a = set(range(0, N, 2))
count = 0
start = time.time()
for x in range(N):
    if x in a:
        count += 1
end = time.time()
print(f'elapsed time: {end-start:.6f} sec'))
```

Dictionaries

Dictionaries

- ~~Unordered~~ (Ordered since 3.7) collections of arbitrary objects
- Store key-value pairs: accessed by key, not offset
- Variable-length, heterogeneous, and arbitrarily nestable
- Python's most powerful data structure

```
menu = {'spam':9.99, 'egg':0.99}
a = {1:'a', 1:'b', 2:'a'}
b = {'food':{'ham':1, 'egg':2}}
c = {'food':['spam', 'ham', 'egg']}
emptydict = {}                # dict()
```

Lists vs. Dictionaries

- Dictionaries are like lists except that they use keys instead of numbers to look up values

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(1)
>>> print(lst)
[21, 1]
>>> lst[0] = 23
>>> print(lst)
[23, 1]
```

```
>>> dct = dict()
>>> dct['age'] = 21
>>> dct['course'] = 1
>>> print(dct)
{'course': 1, 'age': 21}
>>> dct['age'] = 23
>>> print(dct)
{'course': 1, 'age': 23}
```

Counters with a Dictionary

- One common use of dictionaries is counting how often we see something

```
>>> lastname = dict()
>>> lastname['kim'] = 1
>>> lastname['lee'] = 1
>>> print(lastname)
{'kim': 1, 'lee': 1}
>>> lastname['kim'] = lastname['kim'] + 1
>>> print(lastname)
{'kim': 2, 'lee': 1}
```

Dictionary Tracebacks

- It is an error to reference a key which is not in the dictionary
- We can use the in operator to see if a key is in the dictionary

```
>>> lastname = dict()
>>> lastname['kim'] = 1
>>> print(lastname['park'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'park'
>>> 'kim' in lastname
True
>>> 'park' in lastname
False
```

Counting with the `in` Operator

- When we encounter a new name, we need to add a new entry in the dictionary
- If this is the second or later time we have seen the name, we simply add one to the count in the dictionary under that name

```
counts = dict()
names = ['kim', 'lee', 'park', 'kim', 'park', 'jang']
for name in names:
    if name not in counts:
        counts[name] = 1
    else:
        counts[name] = counts[name] + 1
print(counts)
```

Counting with get()

- *dict*.get(*key*, [*default*])
  - Return the value of *key* if *key* is in the dictionary, else *default*
  - If *default* is not given, it defaults to None
  - Never raises a KeyError

```
counts = dict()
names = ['kim', 'lee', 'park', 'kim', 'park', 'jang']
for name in names:
    counts[name] = counts.get(name, 0) + 1
print(counts)
```

```
{'kim': 2, 'lee': 1, 'park': 2, 'jang': 1}
```

Counting Pattern

- Split the line into words

- Loop through the words

- Use a dictionary to track the count of each word independently

```
counts = dict()
line = input('Enter a line: ')

words = line.split()


print('Words:', words)
print('Counting...')
for word in words:
    counts[word] = counts.get(word, 0) + 1
print(counts)
```

Counting Pattern: Example

```
$ python wordcount.py
Enter a line: the clown ran after the car and the car ran into
the tent and the tent fell down on the clown and the car
Words: ['the', 'clown', 'ran', 'after', 'the', 'car', 'and',
'the', 'car', 'ran', 'into', 'the', 'tent', 'and', 'the',
'tent', 'fell', 'down', 'on', 'the', 'clown', 'and', 'the',
'car']
Counting...
{'the': 7, 'clown': 2, 'ran': 2, 'after': 1, 'car': 3, 'and':
3, 'into': 1, 'tent': 2, 'fell': 1, 'down': 1, 'on': 1}
```

Loops over Dictionaries

- Even though dictionaries are not stored in order, we can write a `for` loop that goes through all the entries in a dictionary
- Actually it goes through all of the keys in the dictionary

```
>>> counts = {'kim': 2, 'lee': 1, 'park': 2, 'jang': 1}
>>> for key in counts:
...     print(key, counts[key])
kim 2
lee 1
park 2
jang 1
```

Retrieving Lists of Keys and Values

- Use *dict*.`keys()`, *dict*.`values()`, and *dict*.`items()`
- You can loop over them!

```
>>> counts = {'kim': 2, 'lee': 1, 'park': 2, 'jang': 1}
>>> print(counts.keys())
dict_keys(['kim', 'lee', 'park', 'jang'])
>>> print(counts.values())
dict_values([2, 1, 2, 1])
>>> print(counts.items())
dict_items([('kim', 2), ('lee', 1), ('park', 2), ('jang', 1)])
>>> total_count = 0
>>> for count in counts.values():
...     total_count += count
```

Looping over *dict*.`items()`

- Loop through the key-value pairs using two iteration variables
- The first variable is the key and the second is the corresponding value

```
>>> counts = {'kim': 2, 'lee': 1, 'park': 2, 'jang': 1}
>>> total_count = 0
>>> for k, v in counts.items():
...     print(k, v)
...     total_count += v
kim 2
lee 1
park 2
jang 1
>>> print(total_count)
6
```

Sorting a Dictionary by Keys

```
>>> d = {'s':4, 'e':1, 'o':9, 'u':0, 'l':3}

>>> for k in sorted(d):
...     print(k, d[k])

>>> for k, v in sorted(d.items()):
...     print(k, v)
```

Sorting a Dictionary by Values

```
>>> d = {'s':4, 'e':1, 'o':9, 'u':0, 'l':3}

>>> for k in sorted(d, key=d.get):
...     print(k, d[k])

>>> for k, v in sorted(d.items(), key=lambda x: x[1]):
...     print(k, v)

>>> for v, k in sorted([(v, k) for k, v in d.items()]):
...     print(k, v)
```

Dictionary Comprehension (1)

```
>>> d = { chr(ord('a')+k):k for k in range(0,5) }
>>> d
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4 }
>>> { k*2:v**2 for k, v in d.items() }
{'aa': 0, 'bb': 1, 'cc': 4, 'dd': 9, 'ee': 16}
>>> { i:f'{i**0.5:.2f}' for i in range(1,10) if i % 2 == 0 }
{2: '1.41', 4: '2.00', 6: '2.45', 8: '2.83'}
>>> tempf = { 'seoul': 20, 'nyc': 10 }
>>> tempc = { k:(5.0 / 9.0)*(v-32) for k, v in tempf.items() }
>>> tempc
{'seoul': -6.666666666666667, 'nyc': -12.222222222222223 }
```

Dictionary Comprehension (2)

```
>>> months = [ 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun' ]
>>> days = [ 31, 28, 31, 30, 31, 30 ]
>>> d2 = { m:d for m, d in zip(months, days) }   # dict(zip(months, days))
>>> d2
{'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30, 'May': 31, 'Jun': 30}
>>> { m:d for m, d in d2.items() if d > 30 }
{'Jan': 31, 'Mar': 31, 'May': 31}
>>> { i:m for i, m in enumerate(months) }
{0: 'Jan', 1: 'Feb', 2: 'Mar', 3: 'Apr', 4: 'May', 5: 'Jun'}
>>> { i:j for i, j in zip(list('ABCDE'), range(5)) }
{'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4}
```

Summary

	String	List	Tuple	Dictionary	Set

Initialization	`r = str()`
	`r = ''`
		`l = list()`
		`l = []`
			`t = tuple()`
			`t = ()`
				`d = dict()`
				`d = {}`
					`s = set()`

Example	`r = '123'`	`l = [1, 2, 3]`	`t = (1, 2, 3)`	`d = {1:'a', 2:'b'}`	`s = {1, 2, 3}`
Category	Sequence	Sequence	Sequence	Collection	Collection
Mutable?	No	Yes	No	Yes	Yes
Items ordered?	Yes	Yes	Yes	~~No~~ (now Yes)	No
Indexing/slicing	Yes	Yes	Yes	No	No
Duplicate items?	Yes	Yes	Yes	No (unique keys)	No
Items sorted?	No	No	No	No	No
`in` operator	Yes	Yes	Yes	Yes	Yes

File I/O

A Text File

- A text file can be thought of as a sequence of lines

```
The First Book of Moses:  Called Genesis

1:1 In the beginning God created the heaven and the earth.
1:2 And the earth was without form, and void; and darkness was upon the face of the deep.
And the Spirit of God moved upon the face of the waters.
1:3 And God said, Let there be light: and there was light.
1:4 And God saw the light, that it was good: and God divided the light from the darkness.
1:5 And God called the light Day, and the darkness he called Night. And the evening and the
morning were the first day.
1:6 And God said, Let there be a firmament in the midst of the waters, and let it divide
the waters from the waters.
1:7 And God made the firmament, and divided the waters which were under the firmament from
the waters which were above the firmament: and it was so.
1:8 And God called the firmament Heaven. And the evening and the morning were the second
day.
1:9 And God said, Let the waters under the heaven be gathered together unto one place, and
let the dry land appear: and it was so.
```

The Newline Character

- We use a special character called the "newline" to indicate when a line ends

- We represent it as \n in strings

- Newline is still one character – not two

```
>>> msg = 'Hello\nWorld!'
>>> msg
'Hello\nWorld!'
>>> print(msg)
Hello
World!
>>> msg = 'X\nY'
>>> print(msg)
X
Y
>>> len(msg)
3
```

File Processing

- A text file has newlines at the end of each line

```
The First Book of Moses:  Called Genesis\n
\n
1:1 In the beginning God created the heaven and the earth.
1:2 And the earth was without form, and void; and darkness was upon the face of the deep.
And the Spirit of God moved upon the face of the waters.\n
1:3 And God said, Let there be light: and there was light.\n
1:4 And God saw the light, that it was good: and God divided the light from the darkness.\n
1:5 And God called the light Day, and the darkness he called Night. And the evening and the
morning were the first day.\n
1:6 And God said, Let there be a firmament in the midst of the waters, and let it divide
the waters from the waters.\n
1:7 And God made the firmament, and divided the waters which were under the firmament from
the waters which were above the firmament: and it was so.\n
1:8 And God called the firmament Heaven. And the evening and the morning were the second
day.\n
1:9 And God said, Let the waters under the heaven be gathered together unto one place, and
let the dry land appear: and it was so.\n
```

Opening a File

- Before we can read the contents of the file, we must tell Python which file we are going to work with and what we will be doing with the file

- This is done with the open( ) function

- open( ) returns a "file handle" – a variable used to perform operations on the file

Using open()

- open(*filename*, *mode*)
  - Creates a Python file object, which serves as a link to a file residing on your machine
  - You can read or write file by calling the returned file object's methods
  - *Filename* is a string (pathname)
  - *mode* is optional: `'r'` to open for text input (default), `'w'` to create and open for text output, `'a'` to open for appending text to the end

```
>>> f = open('genesis.txt')
>>> print(type(f))
<class '_io.TextIOWrapper'>
```

Using with Statement

- File should be closed after use – what if an exception occurs?

```
f = open('genesis.txt', 'w')
f.write('hello, world\n')
f.close()
```

- "with" simplifies file management
  - When you open a file using "with", the file is automatically closed
  - The file is properly closed even if an exception is raised at some point

```
with open('genesis.txt', 'w') as f:
    f.write('hello, world\n')
```

File Handle as a Sequence

- A file handle open for read can be treated as a sequence of strings
- Each line in the file is a string in the sequence
- We can use the for statement to iterate through a sequence

```
with open('genesis.txt') as f:
    for line in f:
        print(line)
```

Counting Lines in a File

- Open a file read-only
- Use a for loop to read each line
- Count the lines and print out the number of lines

```
# open.py
count = 0
with open('genesis.txt') as f:
    for line in f:
        count += 1
print('Line count:', count)


$ python open.py
Line count: 1530
```

Other Ways of Reading Line(s)

- *f*.`readline(`*size=-1*`)`
  - Read and return one line
  - *size*: if specified, at most *size* bytes are read

- *f*.`readlines(`*hint=-1*`)`
  - Read and return a list of lines
  - *hint*: control the number of lines to read

```
with open('genesis.txt') as f:
    while True:
        line = f.readline()
        if not line:
            break
        print(line)
```

Reading the Whole File

- *f*.read(*size*=-1)
  - Read and return at most *size* characters as a single string
  - If *size* is negative or None, read the whole file until EOF

```
>>> f = open('genesis.txt')
>>> contents = f.read()
>>> print(len(contents))
206951
>>> print(contents[:20])
The First Book of Mo
>>> print(contents[-20:])
 a coffin in Egypt.
```

Writing to a File

- *f*.`write(`*s*`)`
  - Write the string `s` to the file and return the number of characters written
  - The file should be open with `'w'` or `'a'`

```
>>> f = open('new.txt', 'w')
>>> f.write('hello, world\n')
13
>>> f.write('Happy New Year %d' % 2021)
20
>>> f.close()
```

When Files are Missing

```
>>> f = open('nofile')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or
directory: 'nofile'
```

Handling Bad File Names

```python
fn = input('Enter a file name: ')
try:
    with open(fn) as f:
        count = 0
        for line in f:
            count += 1
except:
    print('File not found:', fn)
    quit()

print(f'Total {count} lines')
```

```python
fn = input('Enter a file name: ')
try:
    f = open(fn)
except:
    print('File not found:', fn)
    quit()

with f:
    count = 0
    for line in f:
        count += 1
print(f'Total {count} lines')
```

Searching Through a File

- *str*.`startswith()`
  - Put an if statement in our for loop to only print lines that meet some criteria

```
with open('genesis.txt') as f:
    for line in f:
        if line.startswith('1:'):
            print(line)
```

Blank Lines?

- Each line from the file has a newline at the end
- The `print` statement adds a newline to each line

```
1:1 In the beginning God created the heaven and the earth.\n
\n
1:2 And the earth was without form, and void; and darkness was upon the face of the deep. And the Spirit of God moved upon the face of the waters.\n
\n
1:3 And God said, Let there be light: and there was light.\n
\n
1:4 And God saw the light, that it was good: and God divided the light from the darkness.\n
\n
```

Removing the Trailing Newline

- *str*.`rstrip()`
  - Strip the whitespace from the right-hand side of the string
  - Whitespace: blank(`'  '`), tab(`'\t'`), newline(`'\n'`), etc.

```
with open('genesis.txt') as f:
    for line in f:
        if line.startswith('1:'):
            line = line.rstrip()
            print(line)
```

Skipping with Continue

- Skip a line by using the `continue` statement
- *str*.`isdigit()`
  - Return `True` if all characters in the string are digits

```
with open('genesis.txt') as f:
    for line in f:
        if not line[0].isdigit():
            continue
        line = line.rstrip()
        print(line)
```

Using in to Select Lines

- Use an in operator to look for a certain substring in a line

```
with open('genesis.txt') as f:
    for line in f:
        if not line[0].isdigit():
            continue
        if not line.startswith('1:'):
            continue
        if 'heaven' in line:
            line = line.rstrip()
            print(line)
```

Extracting Words

- Use *str*.split()

```
with open('genesis.txt') as f:
    for line in f:
        if not line.startswith('1:'):
            continue
        words = line.strip().split()
        print(words)
```

Finding Top 10 Words

```python
filename = input('Enter file: ')

with open(filename) as f:
    counts = dict()
    for line in f:
        words = line.strip().lower().split()
        for word in words:
            counts[word] = counts.get(word, 0) + 1


for v, k in sorted([(v,k) for k,v in counts.items()], reverse=True)[:10]:
    print(v, k)
```