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

Systems Software &
Architecture Lab.

Seoul National University

Jan. 11 – 15, 2021

Python for Data Analytics

Python
Sequence and Collections

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
  - Lists, strings, tuples

- Collection
  - No ordering
  - Sets, dictionaries

Outline

- Strings
- Files
- Lists
- Tuples
- Sets
- Dictionaries

Strings

String Data Type

- A string is a sequence of characters
- For strings, + means "concatenation" (same as lists)
- When a string contains numbers, it is still a string
- We can convert numbers in a string into a number using `int()`

```
>>> str1 = "Hello"
>>> str2 = 'there'
>>> bob = str1 + str2
>>> print(bob)
Hellothere
>>> str3 = '123'
>>> str3 = str3 + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
TypeError: must be str, not int
>>> x = int(str3) + 1
>>> print(x)
124
```

Slicing Strings

- *str*[*start*:*stop*:*step*]
- Same as list slicing

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> s = 'Monty Python'
>>> print(s[1:2])
o
>>> print(s[8:])
thon
>>> print(s[:])
Month Python
>>> print(s[::2])
MnyPto
```

```
>>> print(s[-4:])
thon
>>> print(s[:-5])
Monty P
>>> print(s[:-6:-1])
nohty
>>> print(s[::-1])
nohtyP ytnoM
```

Strings Have Length

- `len(`*str*`)`
  - The built-in function `len()` gives you the length of a string

b	a	n	a	n	a
0	1	2	3	4	5

- `len()` also works for
  - Lists
  - Tuples
  - Dictionaries
  - Sets
  - …

```
>>> fruit = 'banana'
>>> print(len(fruit))
6
>>> empty = ''
>>> print(len(empty))
0
>>> nl = '\n'
>>> print(len(nl))
1
```

Looping Through Strings

- Using `while` statement

```
fruit = 'banana'
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

- Using `for` statement
  - More elegant (or "Pythonic")

```
fruit = 'banana'
for letter in fruit:
    print(letter)
```

Counting Character(s)

- Loop through each letter in a string and counts the number of times the loop encounters the `'a'` character

```
fruit = 'banana'
count = 0
for letter in fruit:
    if letter == 'a':
        count = count + 1
print(count)
```

- *str*.`count(s)`
  - Return the number of non-overlapping occurrences of substring s

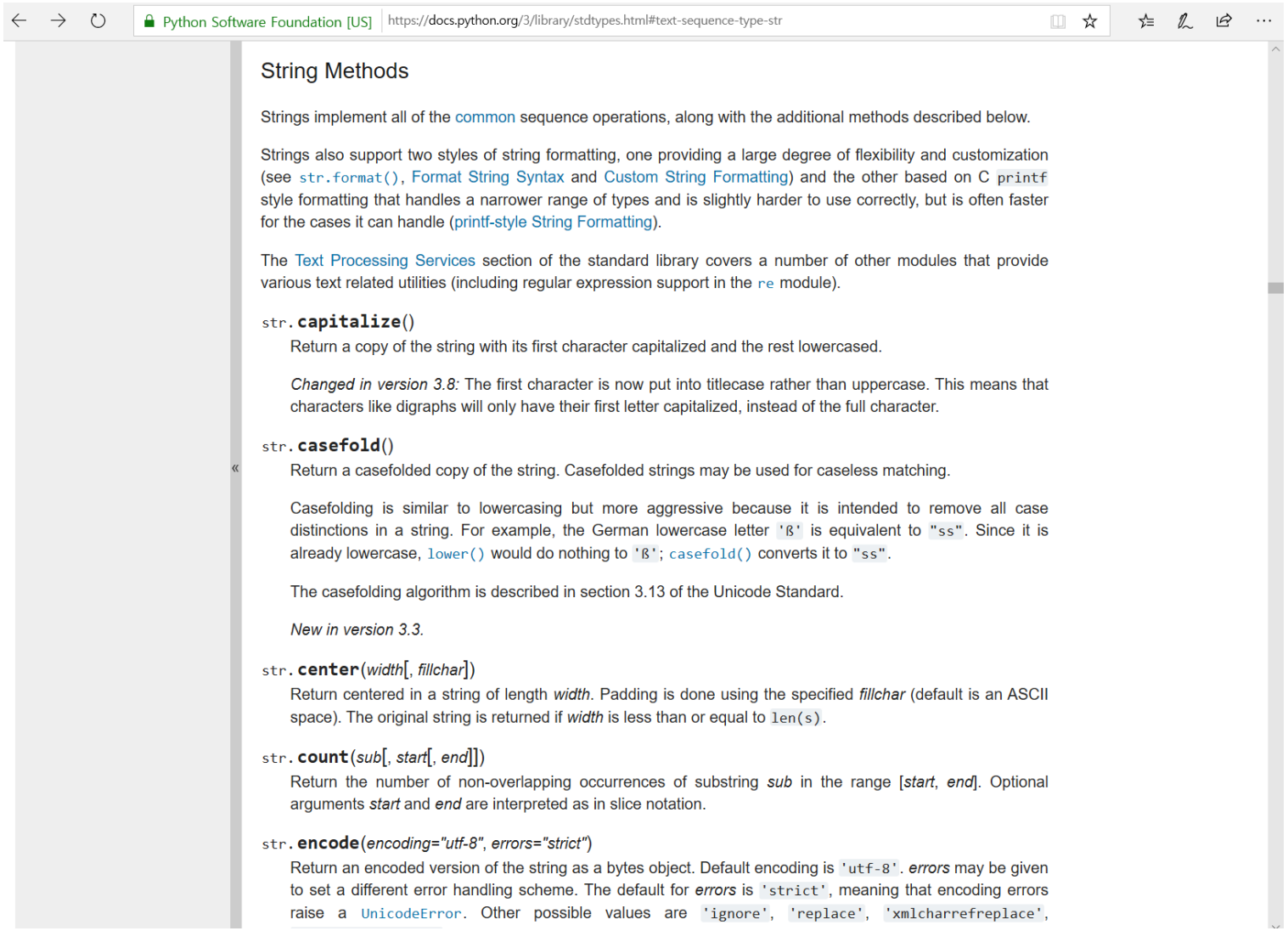
```
fruit = 'banana'
print(fruit.count('a'))
print(fruit.count('na'))
```

The `in` Operator

- Check to see if one string is in another string
- Return `True` or `False`

```
>>> fruit = 'banana'
>>> 'n' in fruit
True
>>> 'm' in fruit
False
>>> 'nan' in fruit
True
>>> if 'a' in fruit:
...     print('Found it!')
Found it!
```

String Methods



String Methods

Strings implement all of the common sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see `str.format()`, Format String Syntax and Custom String Formatting) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle (printf-style String Formatting).

The Text Processing Services section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the `re` module).

`str.`**`capitalize`**()

Return a copy of the string with its first character capitalized and the rest lowercased.

Changed in version 3.8: The first character is now put into titlecase rather than uppercase. This means that characters like digraphs will only have their first letter capitalized, instead of the full character.

`str.`**`casefold`**()

Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter `'ß'` is equivalent to `"ss"`. Since it is already lowercase, `lower()` would do nothing to `'ß'`; `casefold()` converts it to `"ss"`.

The casefolding algorithm is described in section 3.13 of the Unicode Standard.

New in version 3.3.

`str.`**`center`**(*width*[, *fillchar*])

Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.`**`count`**(*sub*[, *start*[, *end*]])

Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

`str.`**`encode`**(*encoding="utf-8"*, *errors="strict"*)

Return an encoded version of the string as a bytes object. Default encoding is `'utf-8'`. *errors* may be given to set a different error handling scheme. The default for *errors* is `'strict'`, meaning that encoding errors raise a `UnicodeError`. Other possible values are `'ignore'`, `'replace'`, `'xmlcharrefreplace'`,

Test

- *str*.`startswith(`*prefix*[,*start*[,*end*]]`)`
  - `True` if string starts with the *prefix*
- *str*.`endswith(`*suffix* [,*start*[,*end*]]`)`
  - `True` if string ends with the suffix
- *str*.`isalpha()`
  - `True` if all characters are alphabetic
- *str*.`isdigit()`
  - `True` if all characters are digits
- *str*.`isprintable()`
  - `True` if all characters are printable
- *str*.`islower()`
  - `True` if all characters are lower case
- *str*.`isupper()`
  - `True` if all characters are uppercase
- *str*.`isspace()`
  - `True` if there are only whitespace characters

Find / Replace

- *str*.`count(`*sub*[,*start*[,*end*]]`)`
- *str*.`find(`*sub*[,*start*[,*end*]]`))`
  - Return the lowest index where substring *sub* is found (-1 if *sub* is not found)
- *str*.`index(`*sub*[,*start*[,*end*]]`))`
  - Like `find()`, but raise `ValueError` if *sub* is not found
- *str*.`replace(`*old*, *new*[,*count*]`))`
  - Return a copy of the string with all occurrences of substring *old* replaced by *new*

```
>>> b = 'banana'
>>> print(b.count('a'))
3

>>> print(b.find('x'))
-1

>>> print(b.index('na'))
2

>>> print(b.replace('a','x'))
bxnxnx
```

Reformat (1)

- *str*.`lower()`
  - Return a copy of the string with all the characters converted to lowercase
- *str*.`upper()`
  - Return a copy of the string with all the characters converted to uppercase
- *str*.`capitalize()`
  - Return a copy of the string with its first character capitalized and the rest lowercased

```
>>> s = 'MoNtY PyThOn'

>>> print(s.lower())
monty python

>>> print(s.upper())
MONTY PYTHON

>>> print(s.capitalize())
Monty python
```

Reformat (2)

- *str*.`lstrip`([*chars*])
  - Return a copy of the string with leading characters removed.
  - If omitted, the *chars* argument defaults to whitespace characters
- *str*.`rstrip`([*chars*])
  - Like `lstrip()`, but trailing characters are removed
- *str*.`strip`([*chars*])
  - *str*.`lstrip`([*chars*]) + *str*.`rstrip`([*chars*])

```
>>> s = '-- monty python --'

>>> print(s.lstrip(' -'))
monty python --

>>> print(s.rstrip('- '))
--- monty python

>>> print(s.strip(' -mno'))
ty pyth
```

Split

- *str*.`split`(*sep*, *maxsplit*)
  - Return a list of the words in the string, using *sep* as the delimiter string
  - If *maxsplit* is given, at most *maxsplit* splits are done. Otherwise all possible splits are made
  - The *sep* argument may consist of multiple characters (None = whitespaces)
  - If *sep* is given, consecutive delimiters are NOT grouped together

```
>>> s = 'hi hello world'
>>> s.split()
['hi', 'hello', 'world']
>>> t = '1:2:3'
>>> t.split(':')
['1', '2', '3']
>>> t.split(':', 1)
['1', '2:3']
>>> t = '1:2::3'
>>> t.split(':')
['1', '2', '', '3']
>>> t.split('::')
['1:2', '3']
```

Join

- *str*.`join(`*iterable*`)`
  - Return a string which is the concatenation of the strings in *iterable*
  - *iterable*: List, Tuple, String, Dictionary, Set
  - The separator between elements is the string (*str*) providing this method

```
>>> menu = ['spam', 'ham', 'egg']
>>> ','.join(menu)
'spam,ham,egg'
>>> ' '.join(menu)
'spam ham egg'
>>> ' * '.join(menu)
'spam * ham * egg'
>>> '#'.join('spam')
's#p#a#m'
```

String and List: Example

```
>>> s = 'spam'
>>> l = list(a)
>>> l
['s', 'p', 'a', 'm']
>>> l[3] = 'n'                  // string is immutable, but list is mutable
>>> l
['s', 'p', 'a', 'n']
>>> t = ''.join(l)
>>> t
'span'
```

String Formatting

- String formatting expression: `'... %s ...' % (values)`

```
>>> print('I have %s.' % 'apples')
I have apples.
>>> print('I have %d %s.' % (2, 'apples'))
I have 2 apples.
```

- String formatting method calls: `'... {} ...'.format(values)`

```
>>> print('I have {}.'.format('apples'))
I have apples.
>>> print('I have {} {}.'.format(2, 'apples'))
I have 2 apples.
```

Formatting Type Code

- %[*flags*][*width*][.*precision*]*typecode*
  - *flags*
    - Left justification (–)
    - Numeric sign (+)
    - Zero fills (0)
    - …
  - *width*: a total minimum field width for the substitute text
  - *precision*: the number of digits to display after a decimal point for floating-point numbers

Code	Meaning
%s	String (or most objects with str())
%c	Character
%d	Integer
%o	Octal integer
%x	Hexadecimal integer
%X	Same as %x, but with uppercase letters
%f	Floating-point decimal
%e	Floating point with exponent, lowercase
%E	Floating point with exponent, uppercase
%%	Literal %

Formatting Expression Examples

```
>>> x = 1234
>>> s = 'integers: ...%d...%-6d...%06d' % (x, x, x)
>>> print(s)
integers: ...1234...1234  ...001234

>>> f = 3.14159265
>>> print('%e | %E | %f' % (f, f, f))
3.141593e+00 | 3.141593E+00 | 3.141593
>>> print('%-6.2f | %05.2f | %+06.1f' % (f, f, f))
3.14   | 03.14 | +003.1

>>> h = '0x%08x' % x
>>> print(hex(x), h)
0x4d2 0x000004d2
```

Formatting Method Examples

```
>>> print('{}, {} and {}'.format('spam', 'ham', 'eggs'))
spam, ham and eggs
>>> print('{0}, {2}, and {1}'.format('spam', 'ham', 'eggs'))
spam, eggs, and ham
>>> print('{x}, {y}, and {z}'.format(z='spam', x='ham', y='eggs'))
ham, eggs, and spam
>>> print('{x}, {0}, and {y}'.format('spam', x='ham', y='eggs'))
ham, spam, and eggs

>>> print('{}, {} and {}'.format(-1, 3.14159265, [1, 2, 3, 4]))
-1, 3.14159265 and [1, 2, 3, 4]
>>> x = 3.14156295
>>> print('{0:<10.2f}, {1:>10.5f}, and {val:+10.2f}'.format(x, x, val=x))
3.14      ,    3.14156, and      +3.14
```

Files

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'>
```

File Handle as a Sequence

- A file handle open for read can be treated as a sequence of strings where each line in the file is a string in the sequence
- We can use the `for` statement to iterate through a sequence
- When you open a file using "with", the file is automatically closed

```
f = open('genesis.txt')
for line in f:
    print(line)
f.close()
```

```
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
f = open('genesis.txt')
for line in f:
    count = 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()
```

Searching Through a File

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

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

Removing the Trailing Newline

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

```
f = open('genesis.txt')
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

```
f = open('genesis.txt')
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

```
f = open('genesis.txt')
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()`

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

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

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

count = 0
for line in f:
    count += 1
print('There are', count, 'lines in', fname)
```

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']
```

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
```

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
```

Building a List

- *list*.append(*x*)
  - Add an item 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*-th 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
```

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*
  - Error 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
```

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']
```

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
```

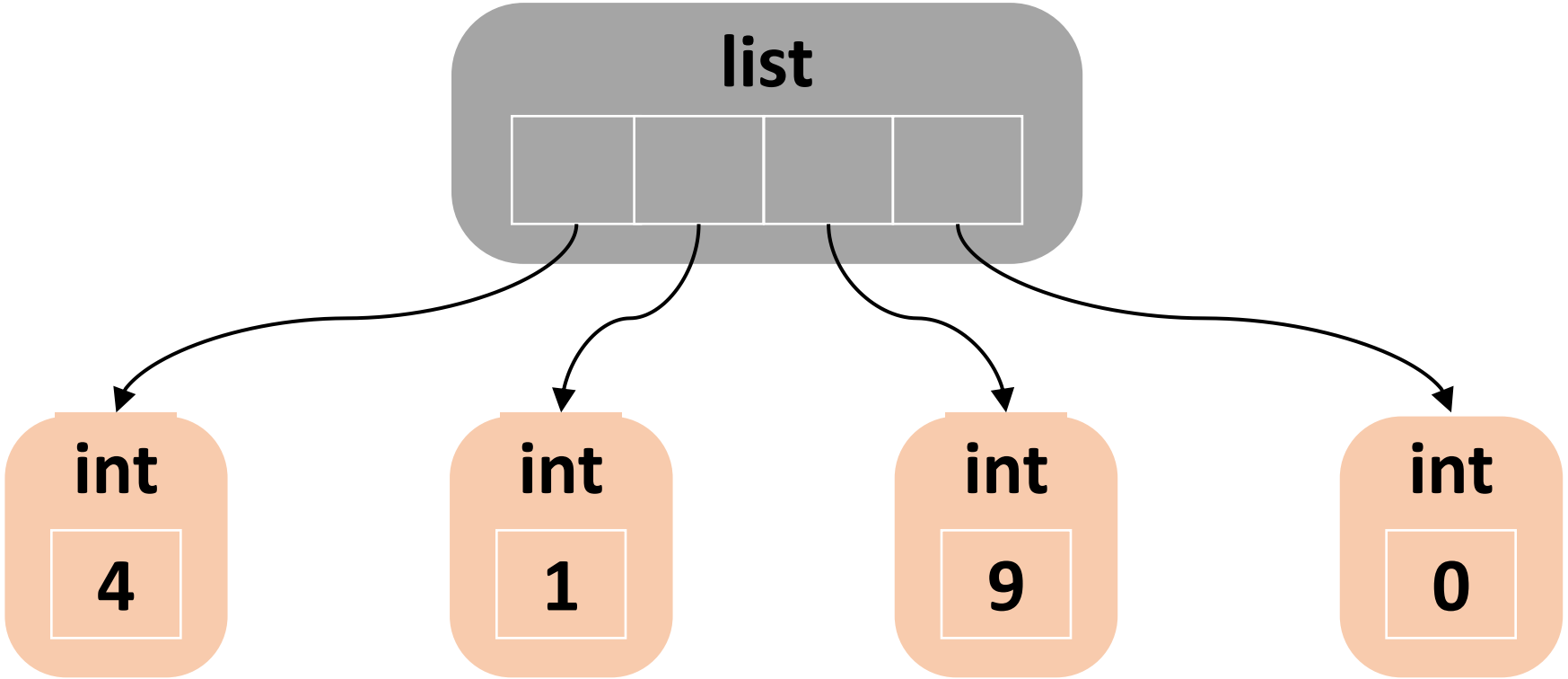
Reversing the List

- *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']
```

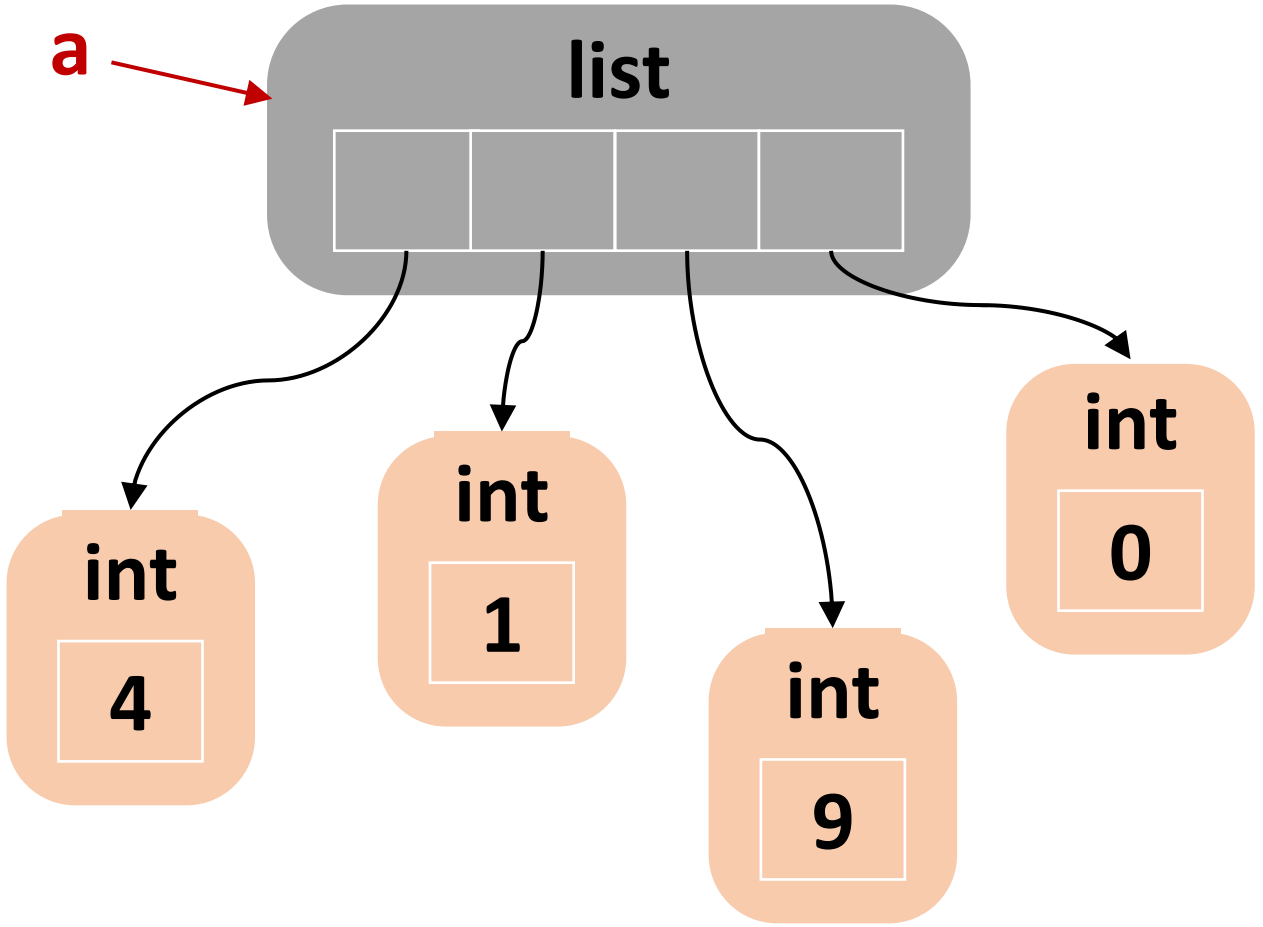
Implementing Lists

- A list has an array of references to other objects



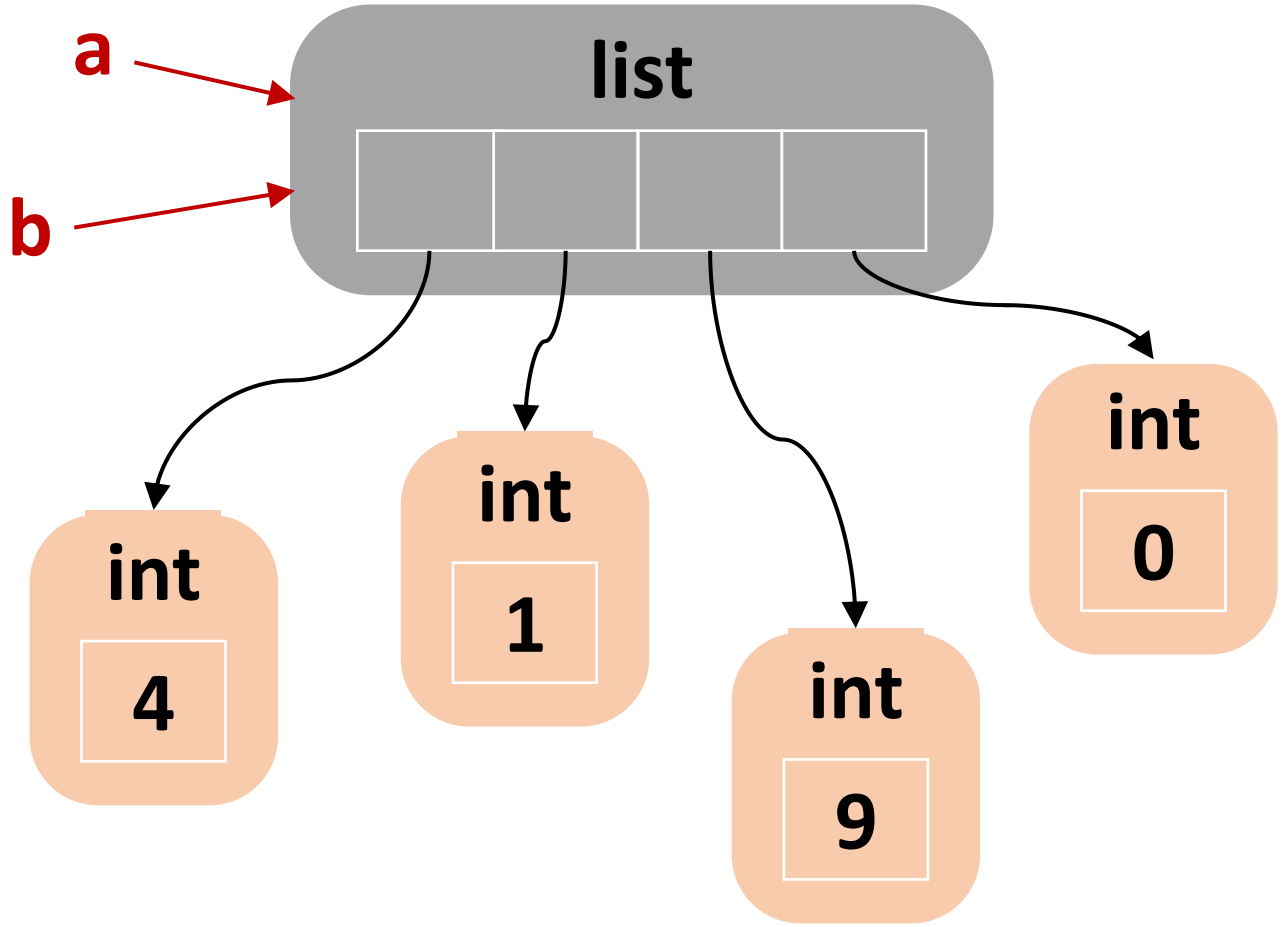
Assigning a List

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



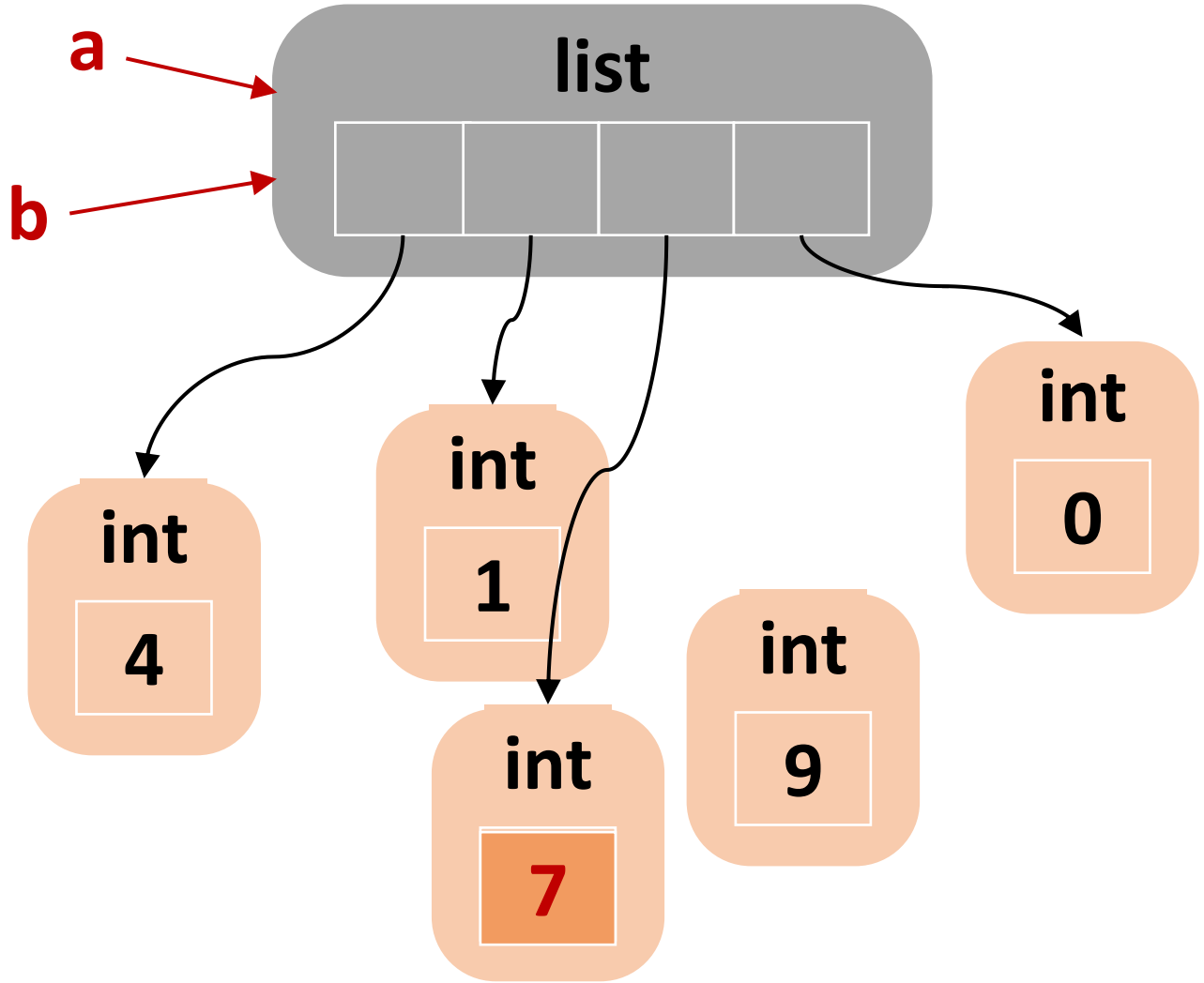
Assigning a List

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



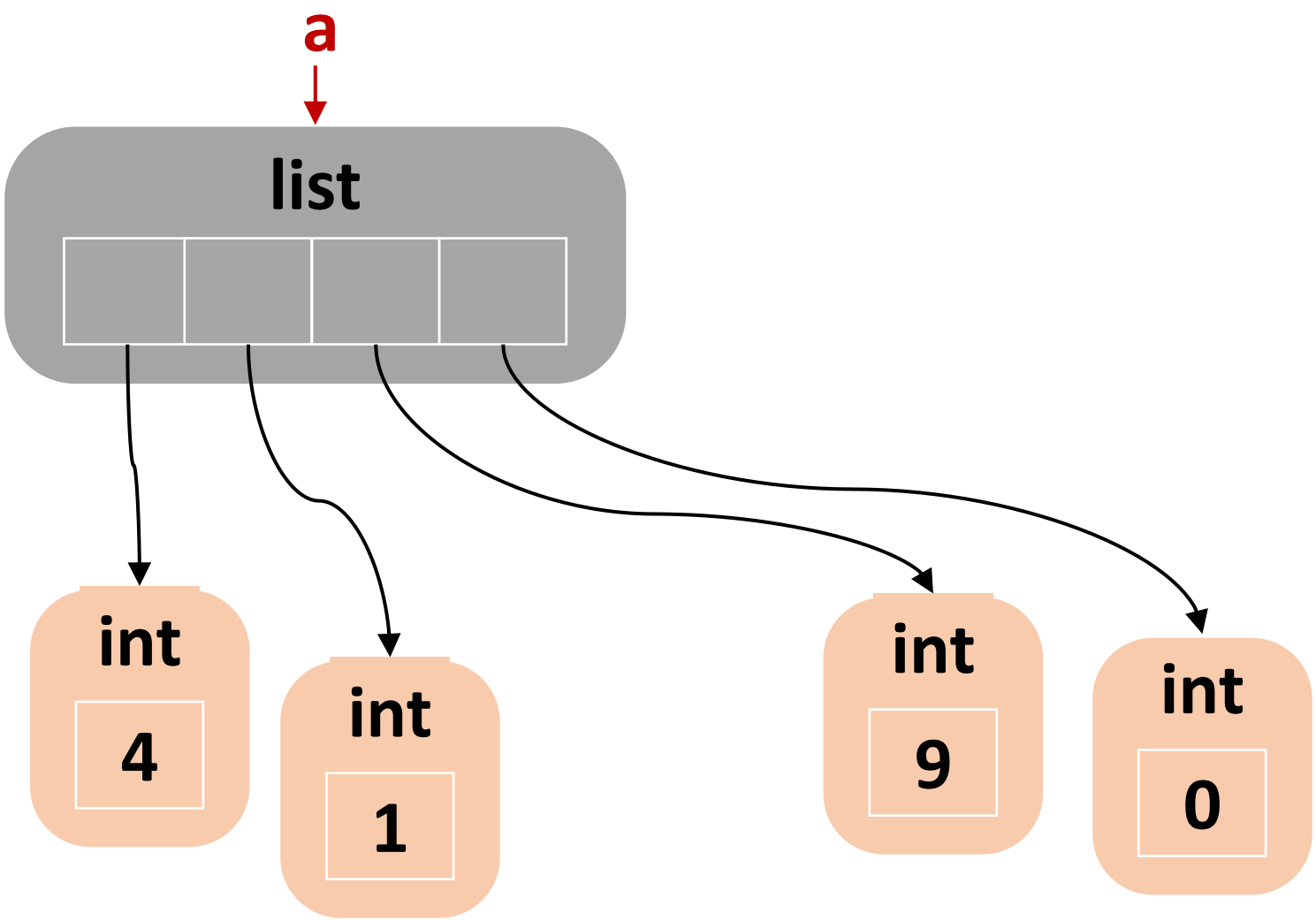
Assigning a List

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



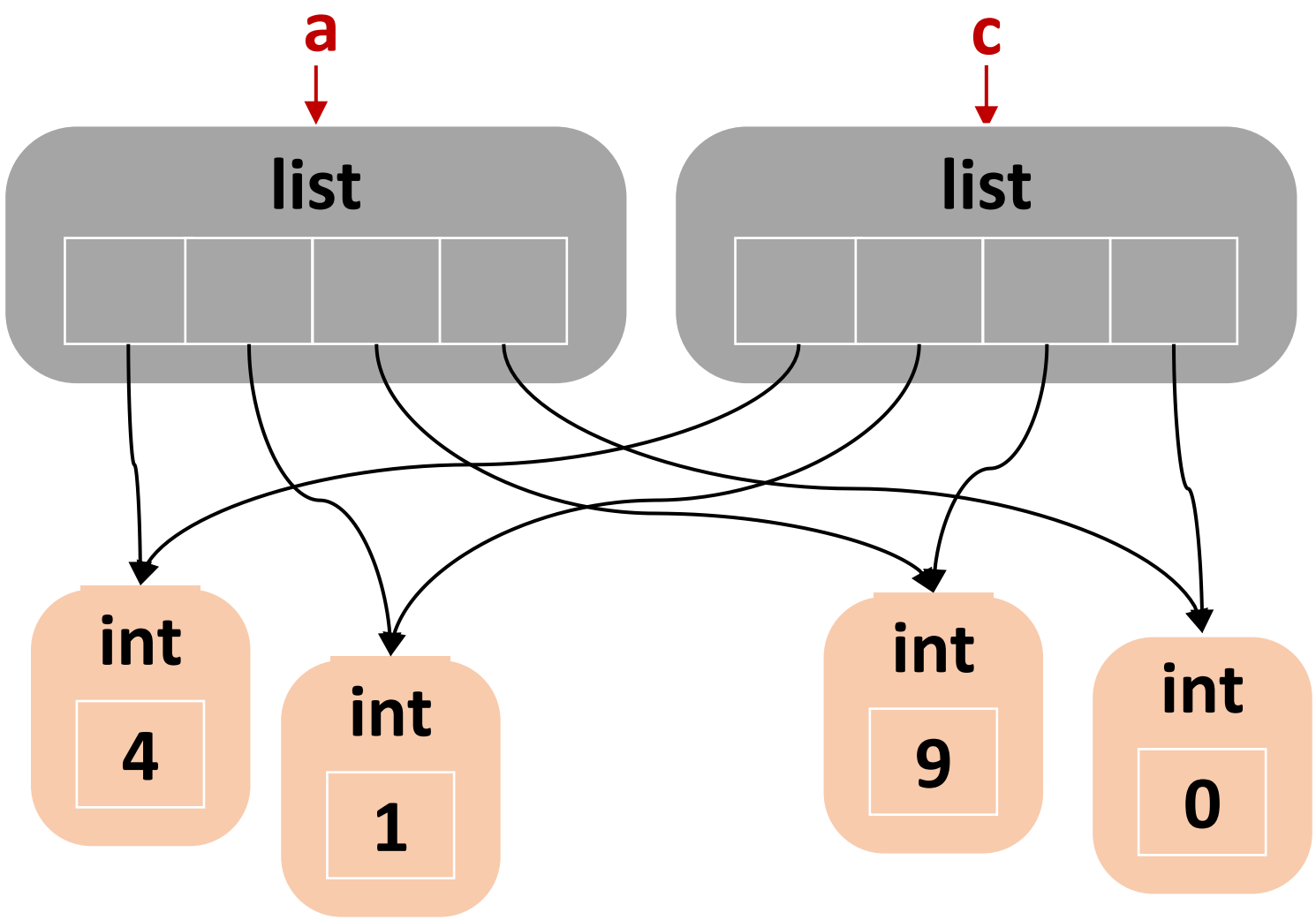
Copying a List

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



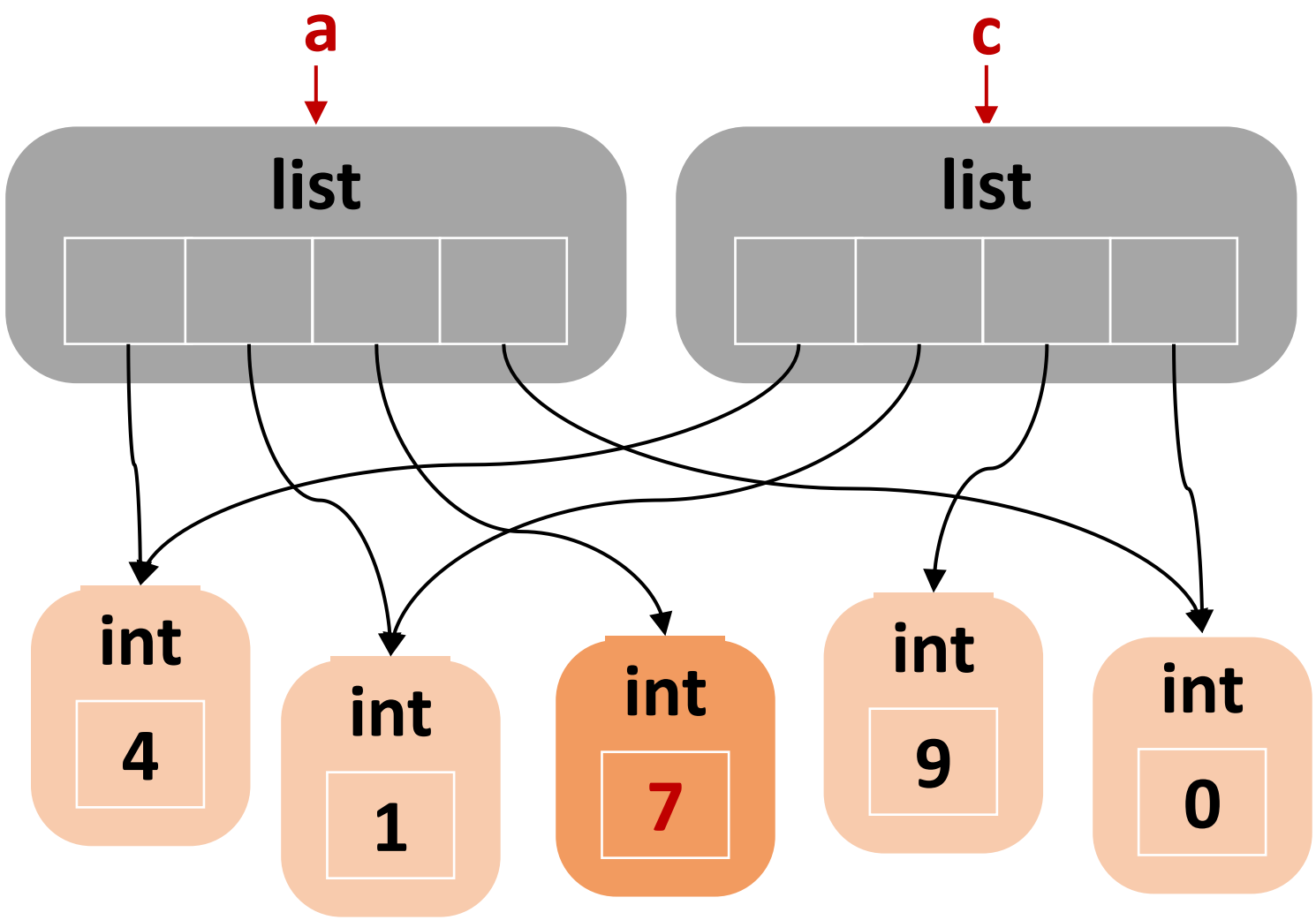
Copying a List

```
>>> a = [4, 1, 9, 0]
>>> c = a.copy()
```



Copying a List

```
>>> a = [4, 1, 9, 0]
>>> c = a.copy()
>>> 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']
```

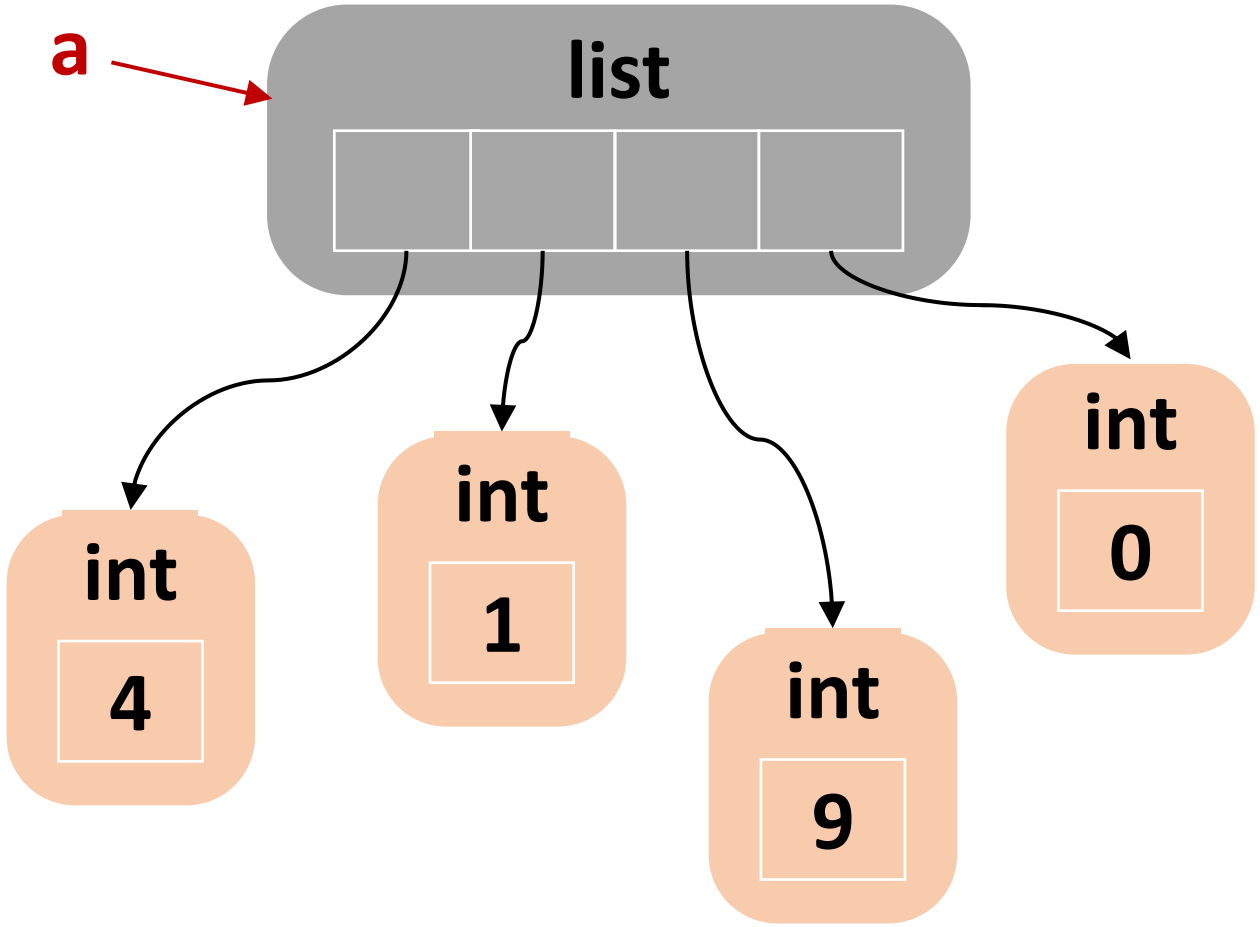
Sorting Elements in a List (2)

- `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']
```

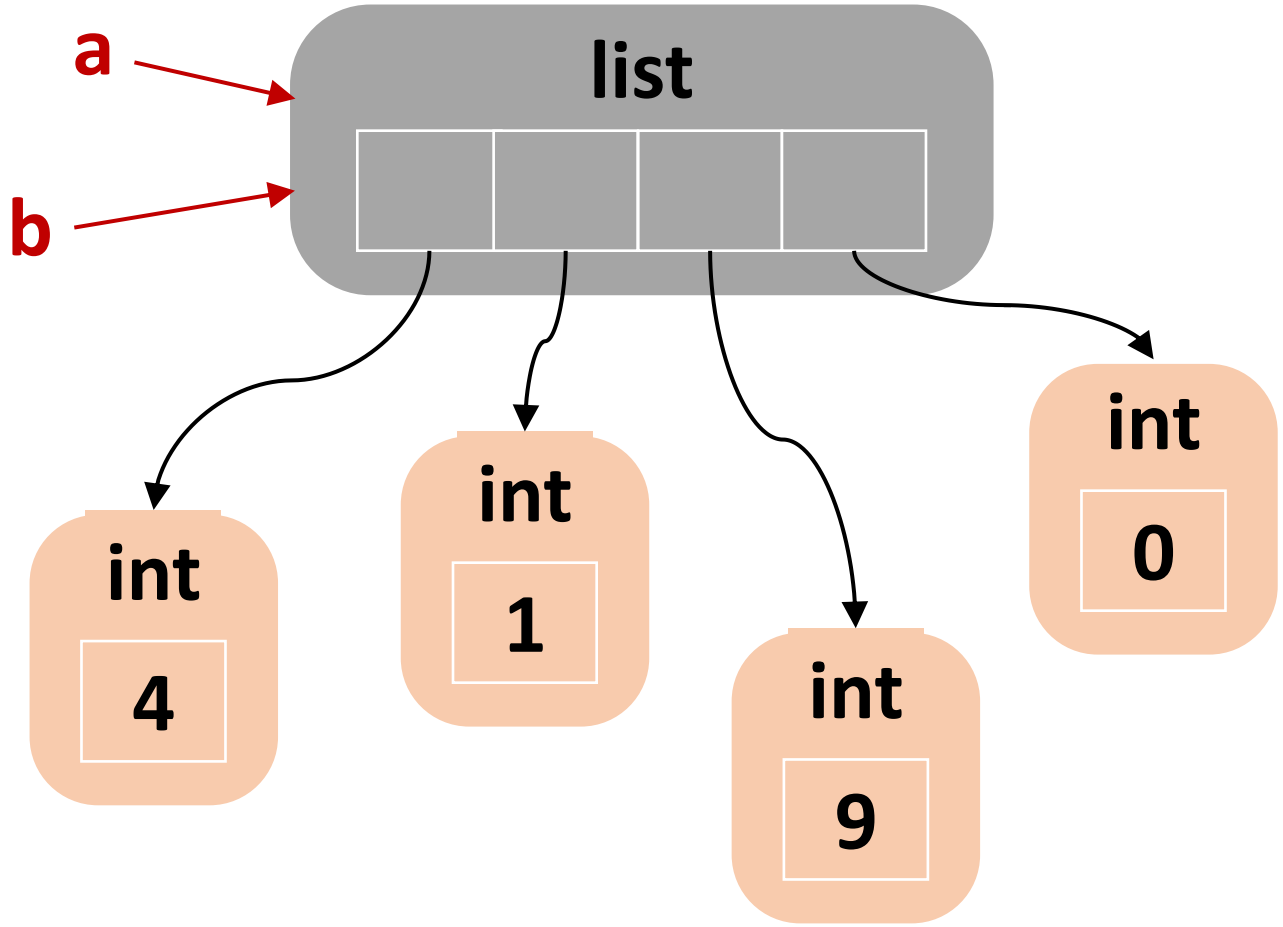
list.sort()

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



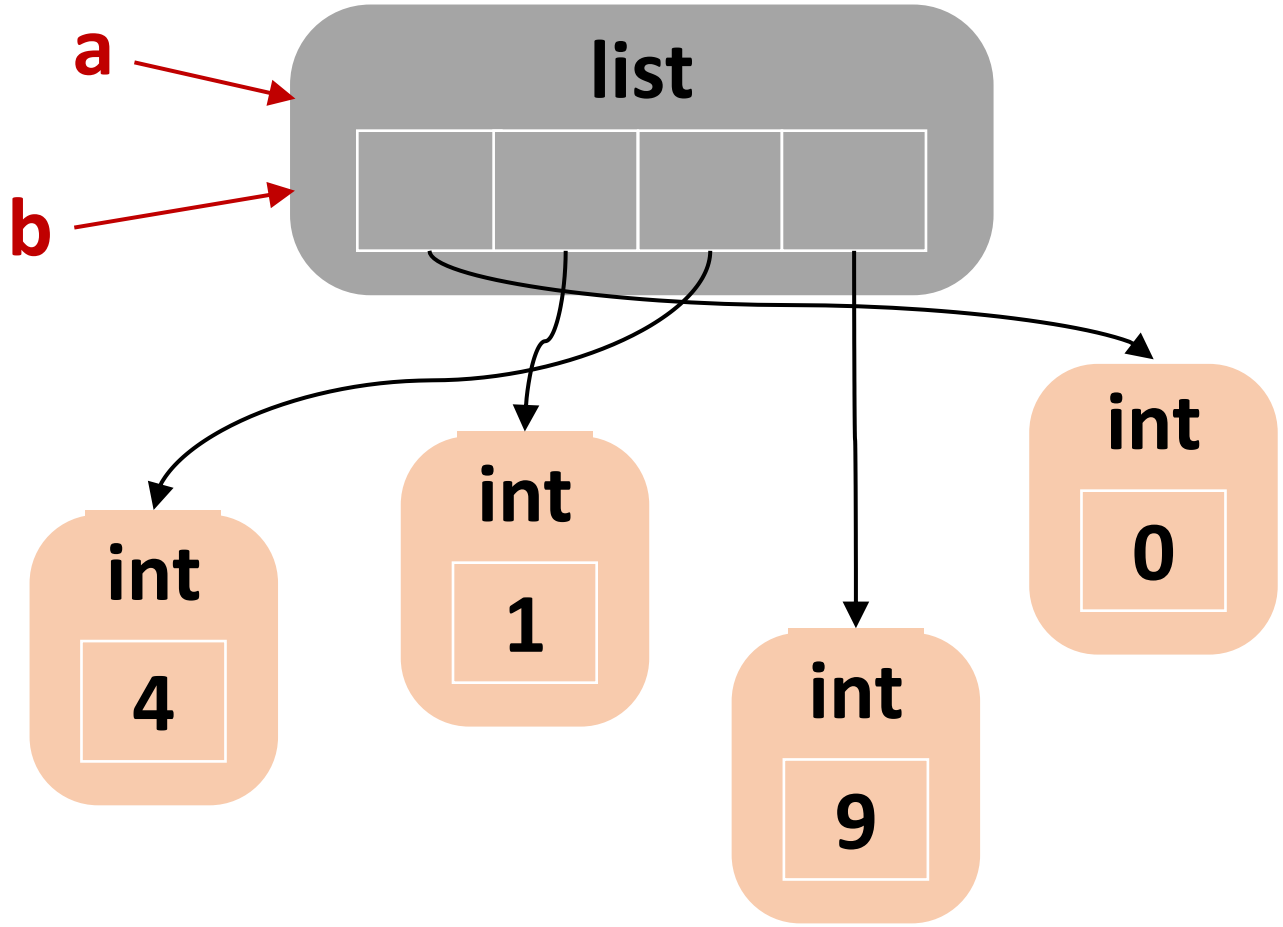
list.sort()

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



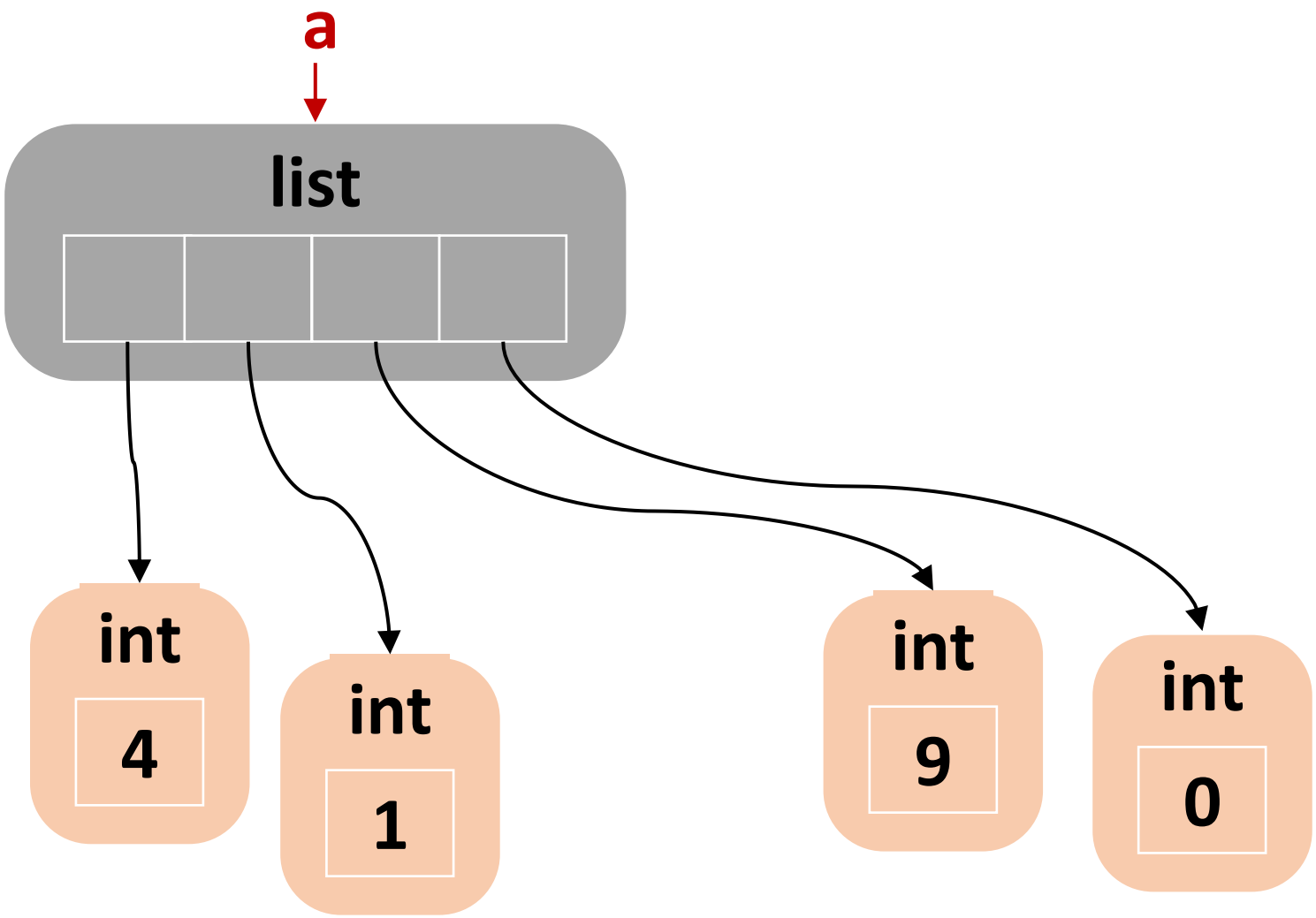
list.sort()

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



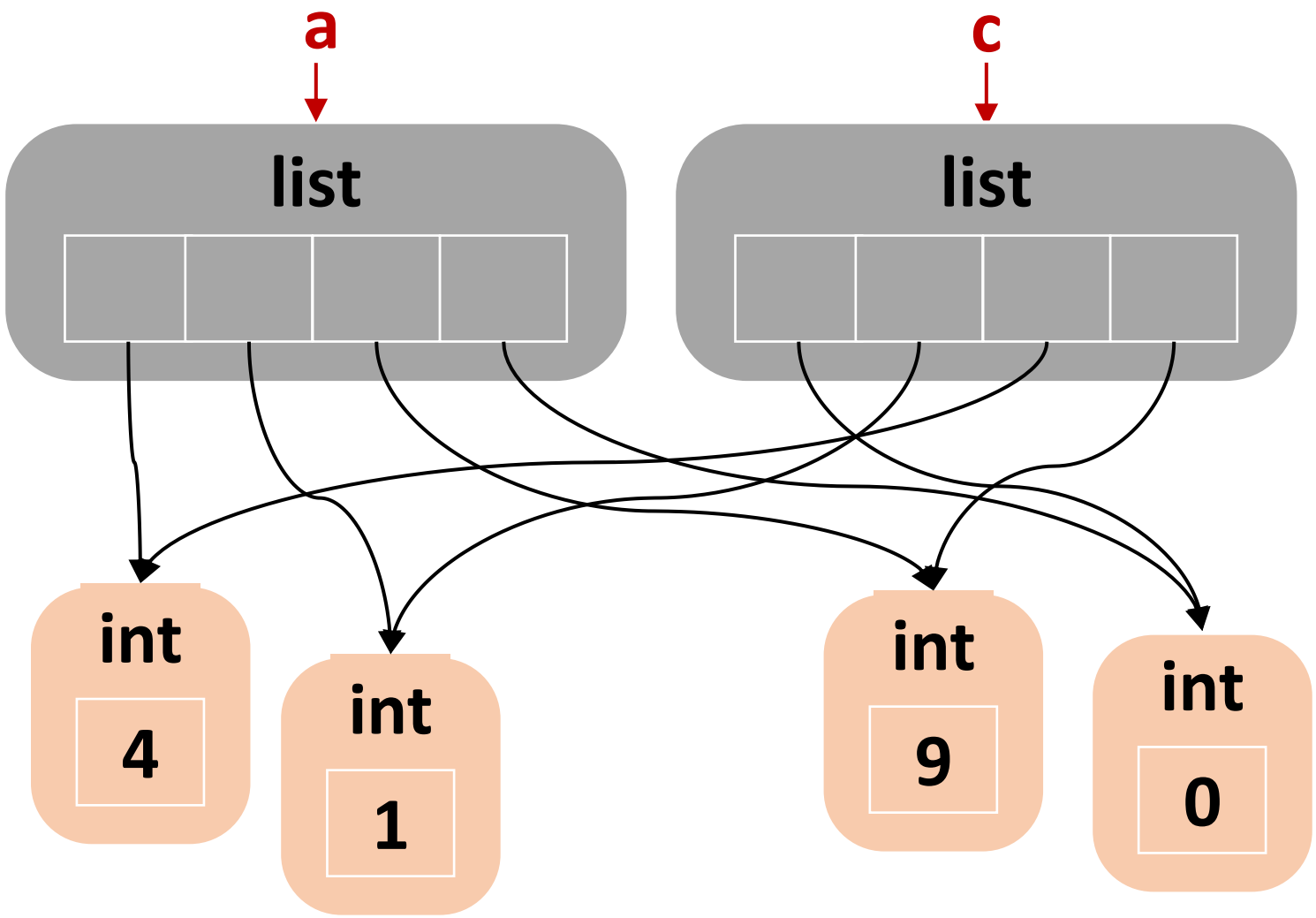
sorted(list)

```
>>> 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]]
```

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) ]
[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]]]
```

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]
```

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 = ()
```

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!

```
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('elapsed time: %.6f sec' % (end - start))
```

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 = {}
```

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)
```

Finding Top 10 Words

```
filename = input('Enter file: ')
f = open(filename)

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

lst = sorted([(v,k) for k,v in counts.items()], reverse=True)

for v, k in lst[:10]:
    print(v, k)
```

```
Enter file: genesis.txt
3630 and
2458 the
1361 of
652 his
644 he
608 to
597 unto
589 in
512 that
470 i
```

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