Ch 8: Implementation of Lists and Sets in Python

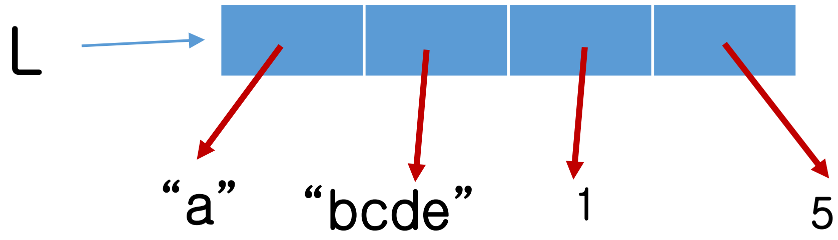
- List Implementation
- Set Implementation

Alternatives of List Implementation in Memory

Consider a list L = [“a”, “bcde”, 1, 5]

Approach A: Array-based Implementation

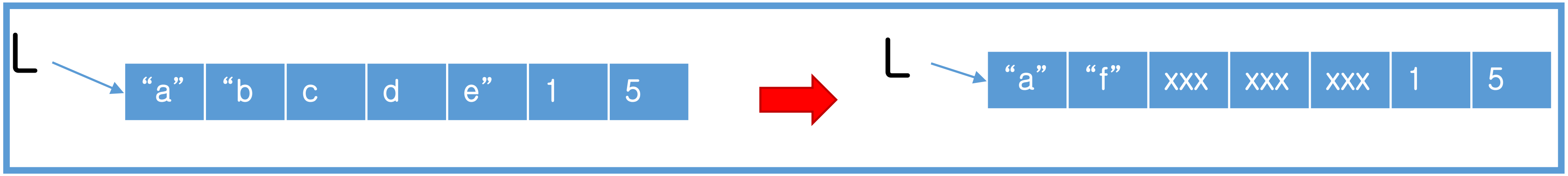
Approach B: Pointer-based Implementation

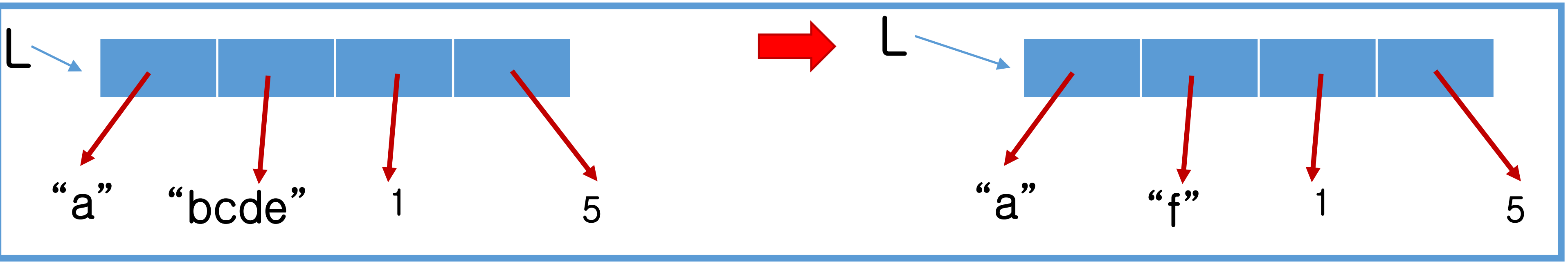


Problem of Array Implementation: Memory Waste

- Consider this modification: L[1] = “f”
  - Then, the 2nd square in our picture will become “f”, and the squares that used to have ‘c’, ‘d, and ‘e’ all become empty and unused.
  - What a waste of memory space!

```
>>> L = [“a”, “bcde”, 1, 5]
>>> L[1] = “f”
```





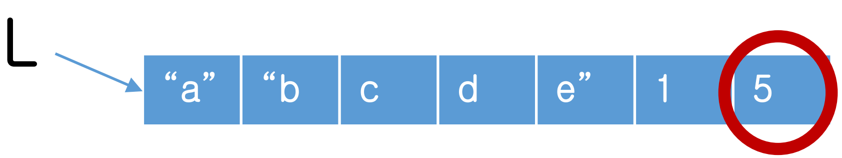
Problem of Array Implementation: Search Inefficiency

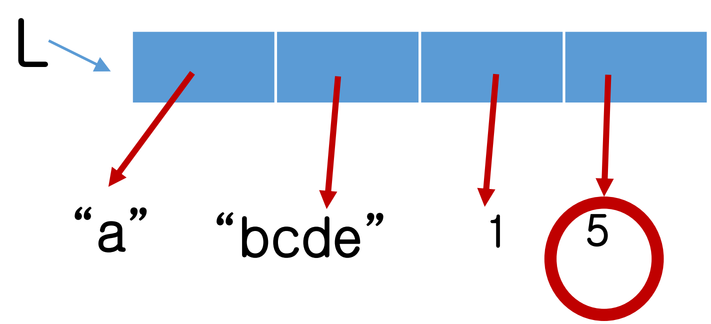
- Consider this search: L[3]

  i.e. We want to check what the element L[3] is.

- But, since some of the elements take up multiple squares, Python must search for more than 4 squares.
  - Then what is the point of indexing a list in the first place?

- However in pointer-based implementation, Python can go to the index 4th (i.e. third) square, and check whatever the pointer in the third square is pointing to.
  - This implementation is what makes indexing valuable

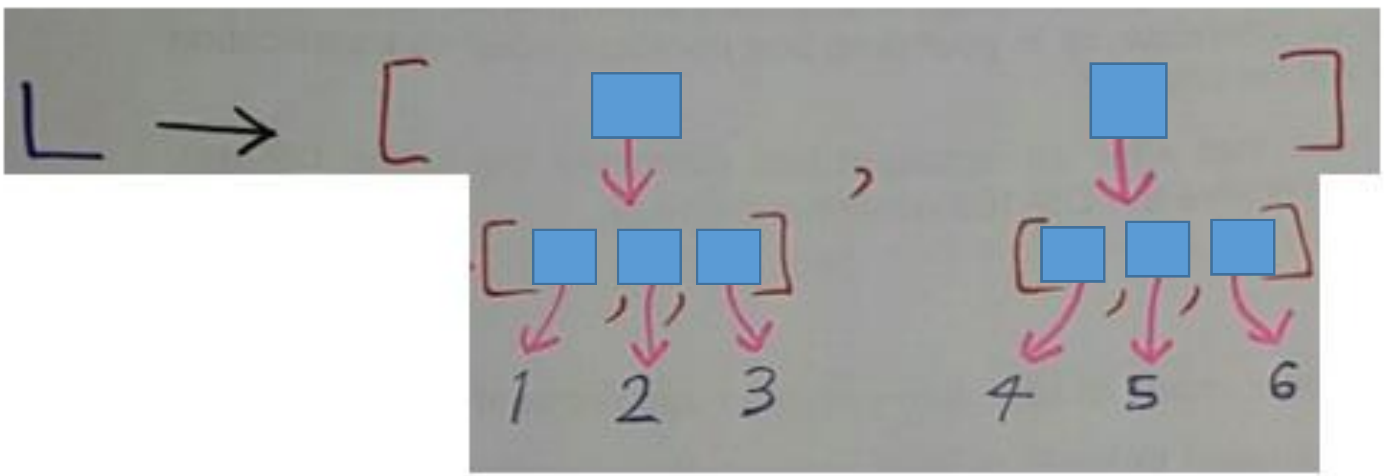




Pointer-based Implementation of 2D List in Memory

```
L = [[1 ,2 ,3], [4, 5, 6]]
```

Inside Memory

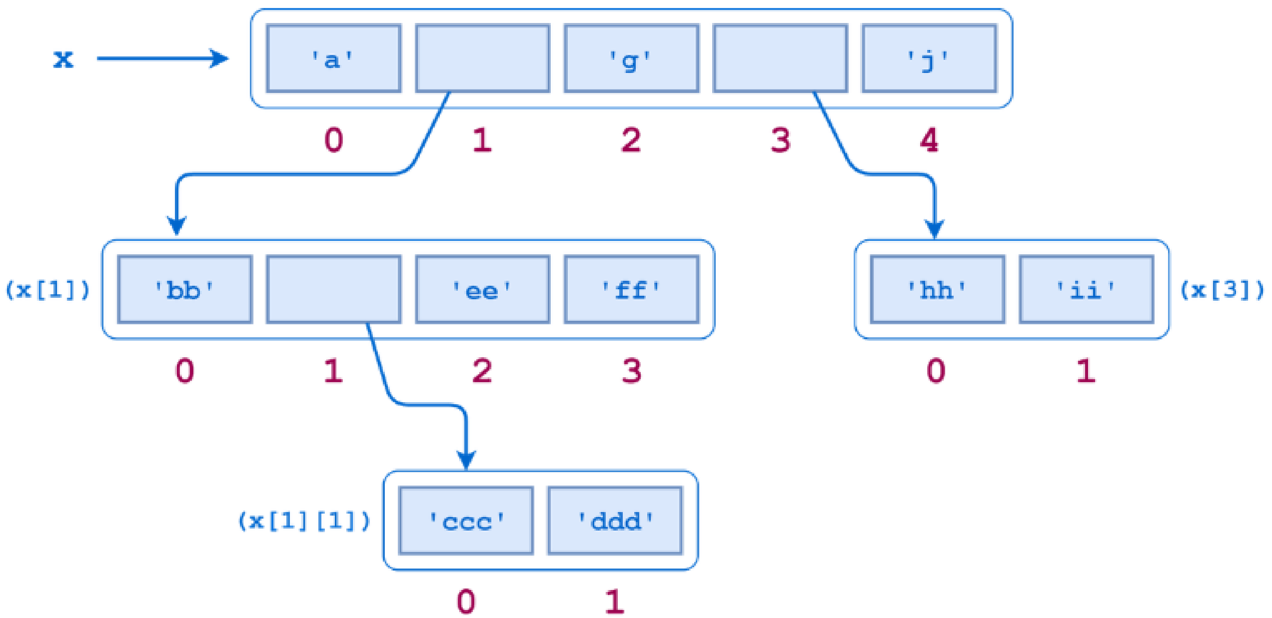


Pointer-based Implementation of 3D List in Memory

Python >>>

```
>>> x = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
>>> x
['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
```

The object structure that x references is diagrammed below:



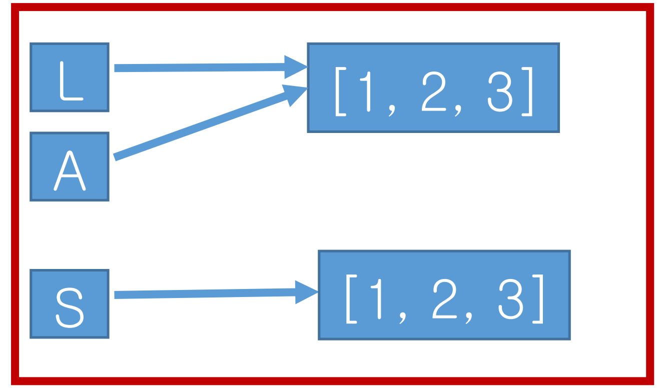
Aliasing, Shallow Copy and Deep Copy in 1D List [1/3]

- Consider the following code

```python
import copy
L = [1, 2, 3]

# A is an alias of L, and S is a copy of L
A = L
S = copy.copy(L)
print("L is: ", L)
print("A is: ", A)
print("S is: ", S)
```

REPL [python]

```
L is:  [1, 2, 3]
A is:  [1, 2, 3]
S is:  [1, 2, 3]
```



- All three lists seem to be the same.
- Now let's modify the lists and see how the other lists are affected.

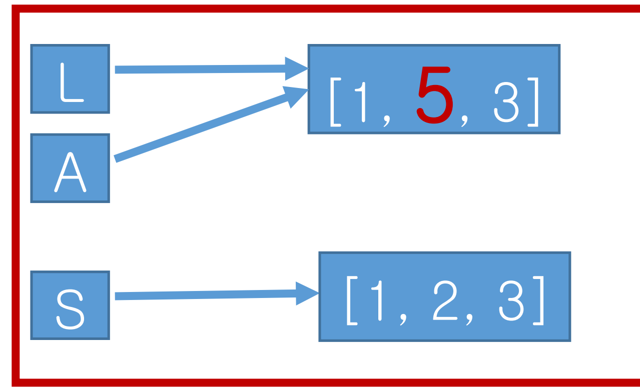
Aliasing, Shallow Copy and Deep Copy in 1D List [2/3]

- If we change the 2nd element (i.e. index 1) of L

```python
L[1] = 5
print("###########")
print("L is ", L)
print("A is ", A)
print("S is ", S)
```

REPL [python]

```
###########
L is  [1, 5, 3]
A is  [1, 5, 3]
S is  [1, 2, 3]
```



The change was applied to both L and A, but not in S (i.e. S was unchanged).

Aliasing, Shallow Copy and Deep Copy in 1D List [3/3]

- When a list is modified, all aliases of that list are modified as well
- But a copy of that list is not affected (unchanged)
- There are many ways to make a copy of a list

    (All these copies are not affected when the original list is modified)

```python
import copy

a = [1, 5, 9]
b = copy.copy(a)
c = copy.deepcopy(a)
d = list(a)
e = a + [ ]
f = a[:]

# change third element (index 2) of a
a[2] = 40
print(a, b, c, d, e, f)
```

REPL [python]

```
[1, 5, 40] [1, 5, 9] [1, 5, 9] [1, 5, 9] [1, 5, 9] [1, 5, 9]
>>>
```

1D List에서는 aliasing과 shallow copy만 차이가 있고

shallow copy나 deep copy가 차이가 없다!

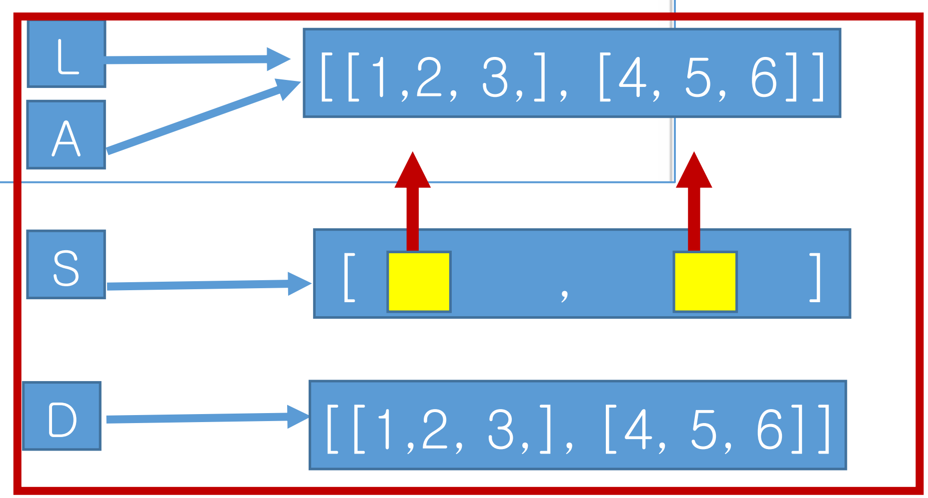
Aliasing, Shallow Copy and Deep Copy in 2D List [1/6]

- Consider the following code

```
import copy

L = [[1 ,2 ,3], [4, 5, 6]]

# A is an alias of L, S is a shallow copy of L, and D is a deepcopy of L
A = L
S = copy.copy(L)
D = copy.deepcopy(L)

print("L is: ", L)
print("A is: ", A)
print("S is: ", S)
print("D is: ", D)
```

REPL [python]

```
L is:  [[1, 2, 3], [4, 5, 6]]
A is:  [[1, 2, 3], [4, 5, 6]]
S is:  [[1, 2, 3], [4, 5, 6]]
D is:  [[1, 2, 3], [4, 5, 6]]
>>>
```



- All four lists seem to be the same.
- Now let's modify the lists and see how the other lists are affected.

Aliasing, Shallow Copy and Deep Copy in 2D List [2/6]

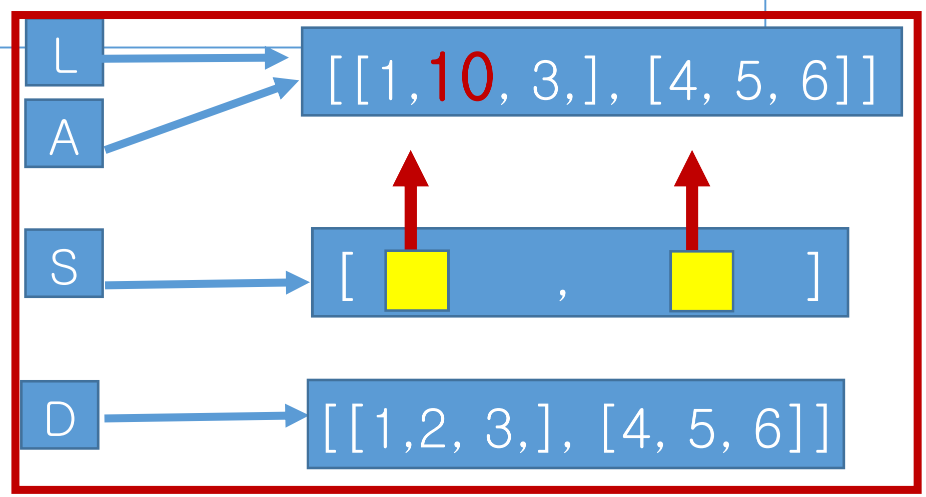
- If we change the 2nd element (i.e. index 1) of the first list in L

```python
# Change the 2nd element (index 1) of the first list in L to 10
L[0][1] = 10
print("L is: ", L)
print("A is: ", A)
print("S is: ", S)
print("D is: ", D)
```

REPL [python]

```
L is:  [[1, 10, 3], [4, 5, 6]]
A is:  [[1, 10, 3], [4, 5, 6]]
S is:  [[1, 10, 3], [4, 5, 6]]
D is:  [[1, 2, 3], [4, 5, 6]]
>>>
```

- L, A (alias), S (shallow copy) were all changed, while D (deep copy) was not affected.



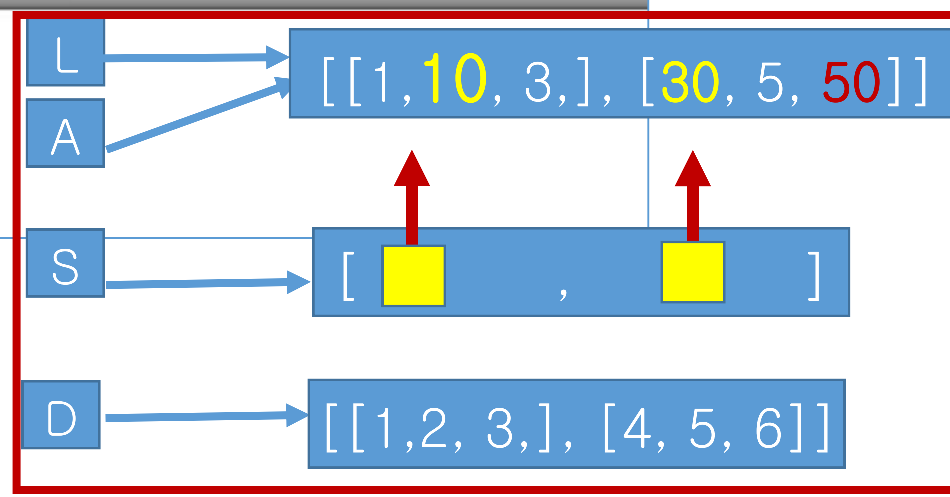
Aliasing, Shallow Copy and Deep Copy in 2D List [3/6]

- If we change the 3rd element (i.e. index 2) of the second list in S (not L)

```python
# Change the 3rd element of the second list in S to 50
S[1][2] = 50
print("L is: ", L)
print("A is: ", A)
print("S is: ", S)
print("D is: ", D)
```

REPL [python]

```
L is:  [[1, 10, 3], [30, 5, 50]]
A is:  [[1, 10, 3], [30, 5, 50]]
S is:  [[1, 10, 3], [30, 5, 50]]
D is:  [[1, 2, 3], [4, 5, 6]]
```



Again, L, A (alias), S (shallow copy) were all changed, while D (deep copy) was not affected.

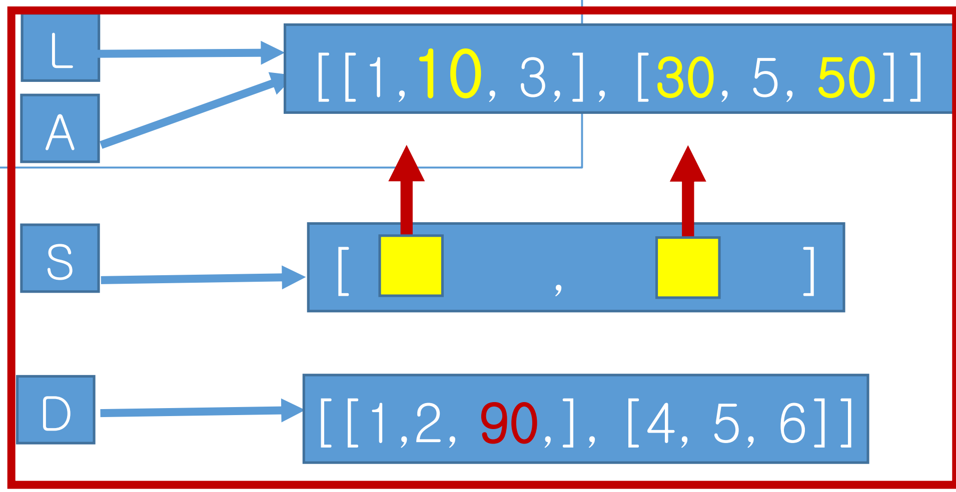
Aliasing, Shallow Copy and Deep Copy in 2D List [4/6]

- If we change the 3rd element (i.e. index 2) of the first list in D (not L)

```python
# Change the 3rd element of the first list in D to 90
D[0][2] = 90
print("L is: ", L)
print("A is: ", A)
print("S is: ", S)
print("D is: ", D)
```

REPL [python]

```
L is:  [[1, 10, 3], [30, 5, 50]]
A is:  [[1, 10, 3], [30, 5, 50]]
S is:  [[1, 10, 3], [30, 5, 50]]
D is:  [[1, 2, 90], [4, 5, 6]]
```



This time, only D (deep copy) was changed, while L, A (alias), S (shallow copy) were all not affected.

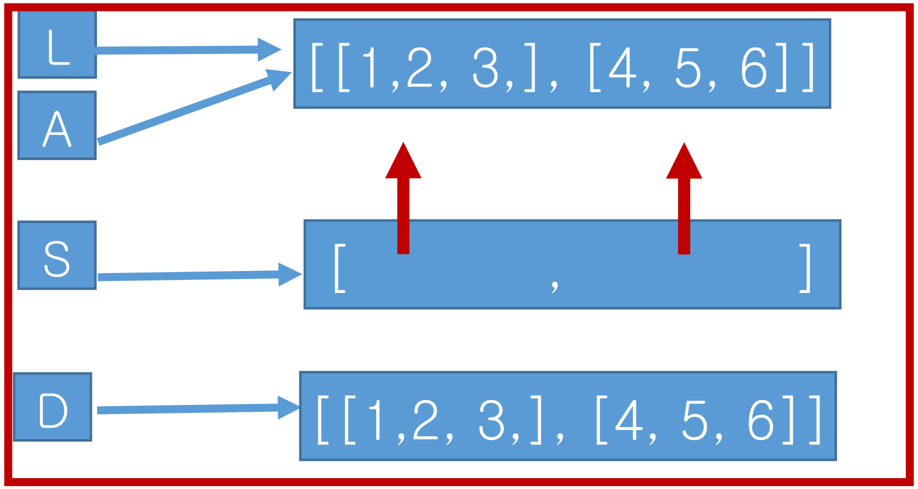
Aliasing, Shallow Copy and Deep Copy in 2D List [5/6]

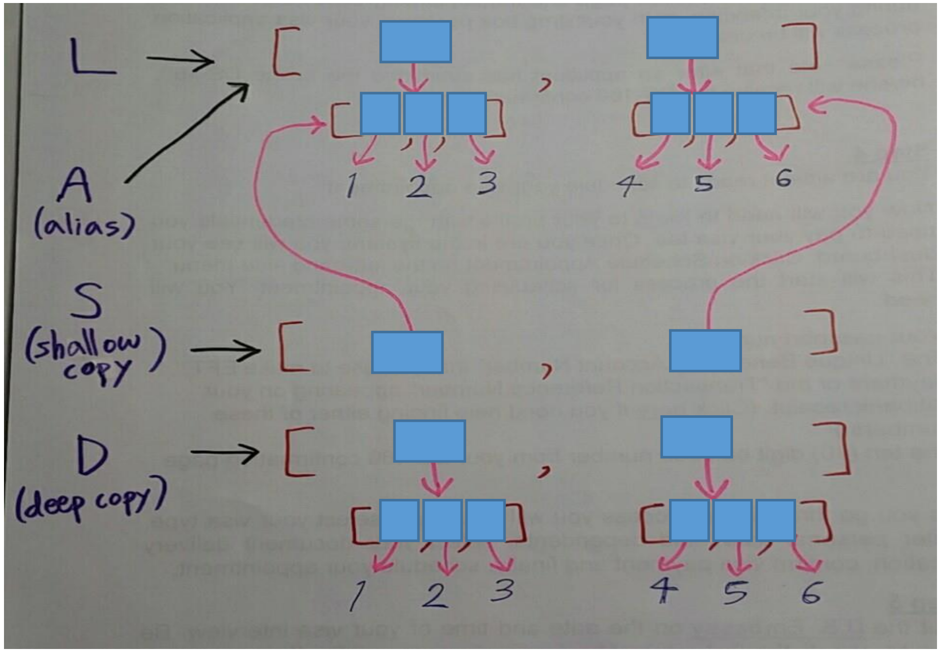
- What really happens in Python when we write this code is…

```
import copy

L = [[1 ,2 ,3], [4, 5, 6]]

# A is an alias of L, S is a
A = L
S = copy.copy(L)
D = copy.deepcopy(L)

print("L is: ", L)
print("A is: ", A)
print("S is: ", S)
print("D is: ", D)
```

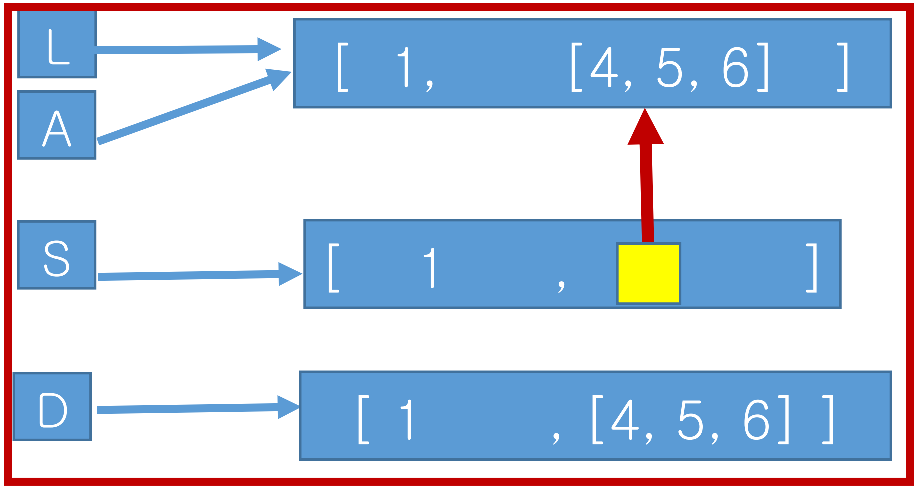
REPL [python]

```
L is:  [[1, 2, 3], [4, 5, 6]]
A is:  [[1, 2, 3], [4, 5, 6]]
S is:  [[1, 2, 3], [4, 5, 6]]
D is:  [[1, 2, 3], [4, 5, 6]]
>>>
```





More Clear Example




```
>>> import copy
>>> L = [ 1, [4, 5, 6]]
>>> A = L
>>> S = copy.copy(L)
>>> D = copy.deepcopy(L)
```

• Suppose

```
>>> A[1][2] = 100
>>> print( "L: ", L , "  A: ", A, "  S: ", S, "  D: ", D)
```

• Suppose

```
>>> S[0] = 9
>>> print( "L: ", L , "  A: ", A, "  S: ", S, "  D: ", D)
```

Aliasing, Shallow Copy and Deep Copy in 2D List [6/6]

In summary, for 2D lists:

- *Aliases* are lists that point to the exact same 2D list
  - Therefore any change in the original 2D list is applied to the alias too

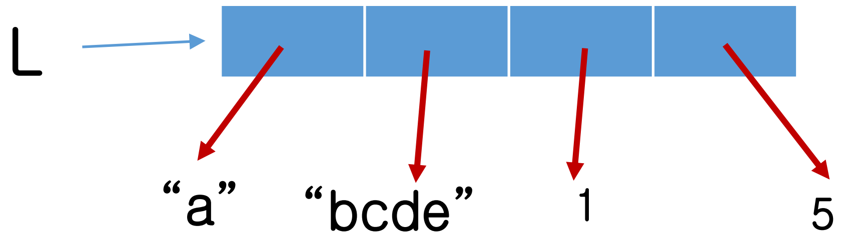
- *Shallow copy* points to a new list, but the elements (which can be 1-d lists) are those of the original 2D lists
  - Therefore change in elements of those inner 1D lists in the original 2D list affect the shallow copy too
  - But adding/removing/changing elements (which can be 1D lists) themselves in the original 2D list does not affect the shallow copy
    - If we add a new 1D list [7, 8, 9] to L, S is unchanged.
    - If we remove [4, 5, 6] from L, S still contains [4, 5, 6]
    - If we change L[1] to [30, 40, 50], S[1] is still [4, 5, 6])

- *Deep copy* points to a new list, where the elements are newly (separately) created identically as those of the original list
  - therefore change in elements of the original 2D list do not affect the deep copy at all

Implementation of Tuple in Memory

Consider a list L = [“a”, “bcde”, 1, 5]

(Pointer-based Implementation)



Flexible
Mutable
Not efficient

Consider a tuple T = (“a”, “bcde”, 1, 5)

(Array-based Implementation)

Not Flexible
Immutable,
Efficient

(Ch 9) Implementation of Lists and Sets in Python

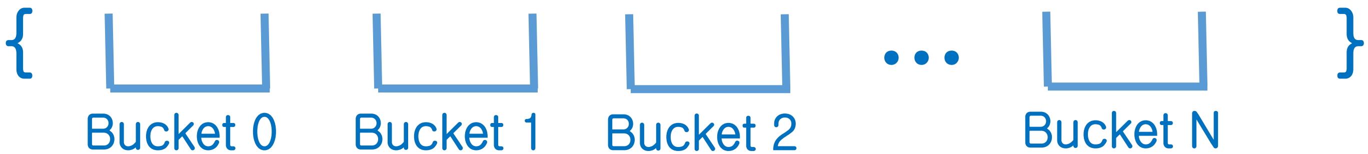
- List Implementation

- Set Implementation
  - How sets are created (Hash)
  - Properties of sets
  - Copies in sets

How Sets are Created [1/2]

- A set is composed of buckets

- S = { 3, 7, 2, 9}



- When we add an element in a set, Python uses **its Hash function H()** to decide which bucket the element will go into

- Therefore, elements of a set should be hashable values

- Immutable values are hashable values
  - Integer, Float, Char/String, Boolean, Tuple are immutable data type
  - List, Set are mutable data type

How Sets are Created [2/2]

- Suppose there are **n-many buckets** in our set
- For an arbitrary element **x** which is **a value of hashable types**, hash(x) returns some number
- Hashable type:
  - int, float, char/string, boolean, tuple, user-defined objects (if user wants)
- Not hashable type: List or Set
  - no set like {[2], [5]} , {{3}, {1,7}}
- It's Python's job to decide the number of buckets, so don't worry about how to get such a number
- Then, Python finds the remainder when that number is divided by the number of buckets. (i.e. hash(x) % n)
- Let's say the solution for hash(x) % n is **y** (so 0 <= y <= n-1)
- Then the element **x** is assigned to bucket number **y**

Properties of Sets

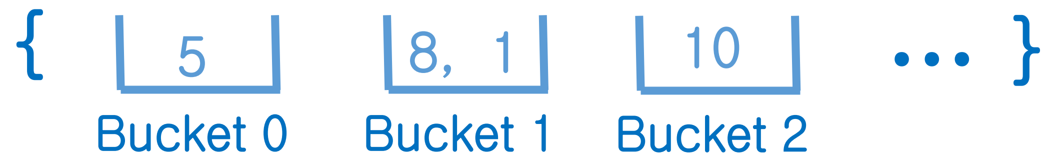
- Sets are unordered collections of values of same data types
- Sets' elements are unique (No repetition)
- Sets' elements must be immutable
- Sets are extremely efficient when finding whether an element is in the set. (The bucket concept is applied here!!)
  - Set containment computation is very fast (element X, set S)
  - (X in S) or (S contains X)

1. Sets are unordered

```python
s = set([1,10,5,8])
print(s)
```

REPL [python]

```
{8, 1, 10, 5}
>>>
```



- As shown in this code, sets' elements are not necessarily ordered in the order the user adds the elements.
  - This is not related to the buckets
  - Python locates each element in the bucket by the algorithm explained previously
  - When we print the set, Python just takes out each element in any order

2. Elements are unique

```python
s = set([1,1,10,10,5,5,8,8])
print(s)
```

REPL [python]

```
{8, 1, 10, 5}
>>>
```

As shown in this code, if there are repetitions of a particular element, then the set only contains 1-many such an element. (No repetition allowed)

3. Elements must be immutable

- Recall that Python hashes each element and assigns the bucket that the element goes into
  - Therefore, if an element is mutable (can change), it will not be consistently hashed into the same bucket
  - Set 안에 있는 element의 위치를 pointing 할수도 없다!

- S = {4, 7, 2} ➔ There is no such S[2] = 100

- But if an element can be hashed into different buckets every time, there is no meaning of using buckets.

- Sets are special because they are so efficient as they use buckets, and this feature is explained in the next slide

4. Sets are more efficient than lists [1/3]

- Set Membership Check : check whether or not a particular element x is in the set s

  (x in s) returns True if x is an element of s and False if not

```python
s = {1,2,3,4}
print(1 in s)
print(10 in s)
```

REPL [python]

```
True
False
>>>
```

4. Sets are more efficient than lists [2/3]

- Set Membership Check :
  (x in s) returns True if x is an element of the set s and False if not
- List Membership Checking:
  (x in L) returns True if x is an element of the list L and False if not

- In a list, Python starts from the beginning and goes through each element to check if it's the same with our element of interest x

- But in a set, Python hashes the given element x and see which bucket x would have been assigned to when the user added x
  - Then it only checks that bucket (and no other element at all) to see if x is in the bucket
  - Therefore, sets are extremely efficient in searching for an element

- Simple Experiment
  - Create a list containing 2, 4, 6, ··· , 29998, 30000
  - Create a set containing 2, 4, 6, ··· , 29998, 30000
  - Then for all x from 0 to 30000 (inclusive), check whether each x is in s and L
  - Let's measure the time for the two cases (for a list and for a set)

4. Sets are more efficient than lists [3/3]

```python
import time

L = []
s = set()
for n in range(2, 30001, 2):
    # even numbers between 2 and 30000 (inclusive)
    L.append(n)
    s.add(n)
# Now, L is a list containing 2, 4, 6, ... , 29998, 30000
# Now, s is a set containing 2, 4, 6, ... , 29998, 30000

############################ LIST ################################
start = time.time()
count = 0
for x in range(30001):
    if x in L:
        count += 1
end = time.time()
listTime = end - start
print("For a list, count =", count," and time = %0.6f seconds" % listTime)
######################### SET ####################################
start = time.time()
count = 0
for x in range(30001):
    if x in s:
        count += 1
end = time.time()
setTime = end - start
print("For a set, count =", count," and time = %0.6f seconds" % setTime)
```

REPL [python]

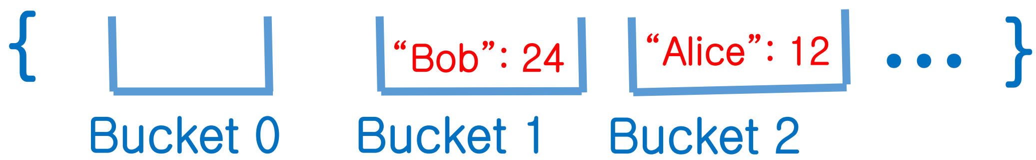
```
For a list, count = 15000  and time = 9.966219 seconds
For a set, count = 15000  and time = 0.015600 seconds
>>>
```

Look at the time difference!
Sets are much more efficient!

Dictionaries

- Dictionaries are similar to sets except that in dictionaries, each element is a <u>pair</u> of a key and a value (the form of key: value)

  e.g. d = {“Alice”: 12, “Bob”: 24}



- In dictionaries, what Python hashes are keys, not values.
- Therefore, keys must be immutable while values may be mutable
  - Basically, keys are elements of sets, so they are unordered, unique, and must be immutable
  - e.g. impossible dictionary

    d = { [2, 3] : 12, [3] : 24 }

    U = { {“kim”, “Lee”}: “SNU”, {“John”, “Tom”}: “Harvard }

Aliasing and Copy of Sets [1/4]

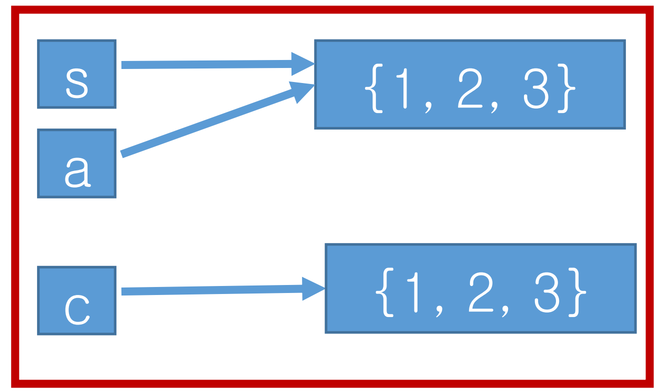
- We can not change the elements of the set (immutable)
  - Set elements are immutable
- We can add or delete the elements of the set
  - Set itself is mutable
- The set can not be nested: i.e., no set like { {1,2}, 4}
  - Therefore, there is no issue of deepcopy in set

SET의 bucket implementation!

```python
import copy
s = {1,2,3}
# a is an alias, and c is a copy of s.
a = s
c = copy.copy(s)
print("Original s is ", s)
print("Alias a is ", a)
print("Copy c is ", c)
```

REPL [python]

```
Original s is  {1, 2, 3}
Alias a is  {1, 2, 3}
Copy c is  {1, 2, 3}
>>>
```



S의 element 2를 9로 변경 : No

S에서 element 2를 제거: Yes
S에 element 9를 삽입: Yes

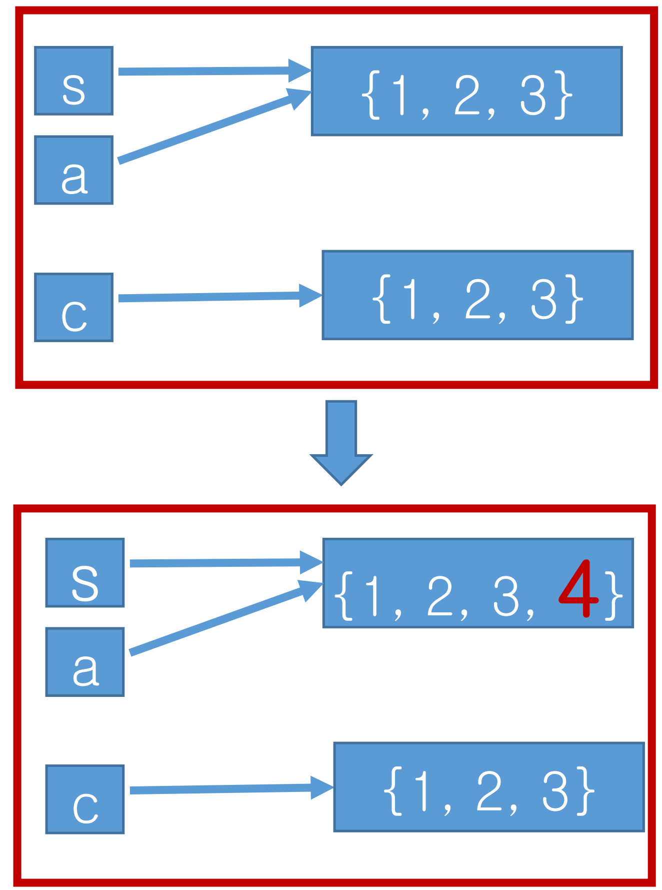
Aliasing and Copy of Sets [2/4]

- If we add 4 in the original set s …

```python
s.add(4)
print("Original s is ", s)
print("Alias a is ", a)
print("Copy c is ", c)
```

REPL [python]

```
Original s is  {1, 2, 3, 4}
Alias a is  {1, 2, 3, 4}
Copy c is  {1, 2, 3}
>>>
```



s and the alias a are affected, but the copy c is not affected.

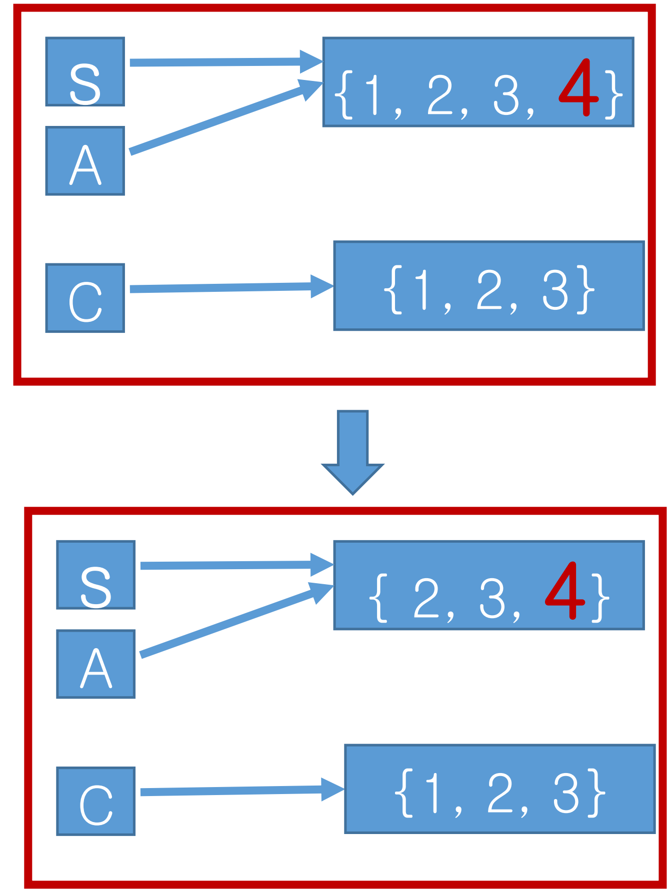
Aliasing and Copy of Sets [3/4]

- If we remove 1 from the alias a …

```python
a.remove(1)
print("Original s is ", s)
print("Alias a is ", a)
print("Copy c is ", c)
```

REPL [python]

```
Original s is  {2, 3, 4}
Alias a is  {2, 3, 4}
Copy c is  {1, 2, 3}
>>>
```



s and the alias a are affected, but the copy c is not affected.

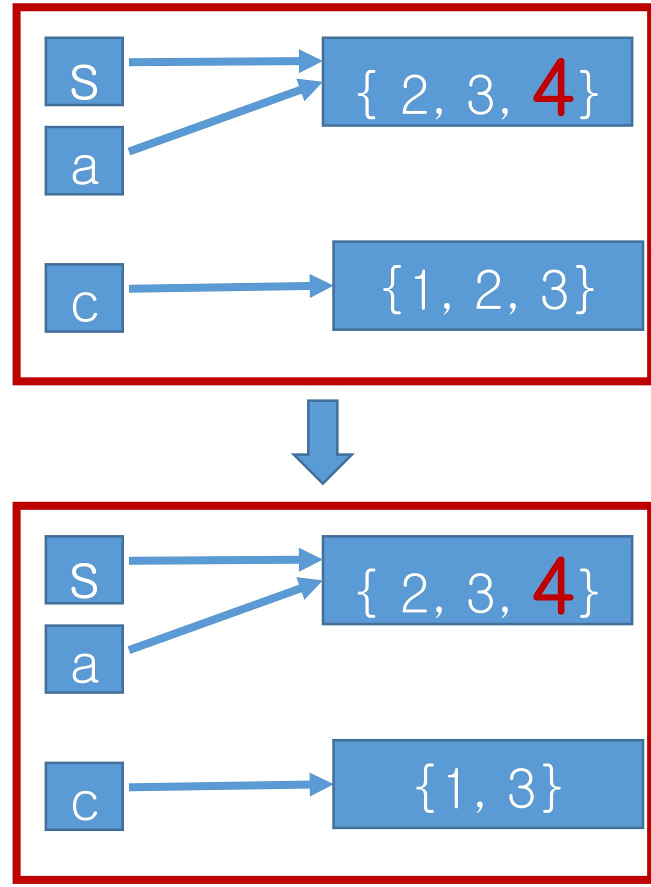
Aliasing and Copy of Sets [4/4]

If we remove 2 from the copy c …

```python
c.remove(2)
print("Original s is ", s)
print("Alias a is ", a)
print("Copy c is ", c)
```

REPL [python]

```
Original s is  {2, 3, 4}
Alias a is  {2, 3, 4}
Copy c is  {1, 3}
>>>
```



The copy c is affected, but s and the alias a are not affected.