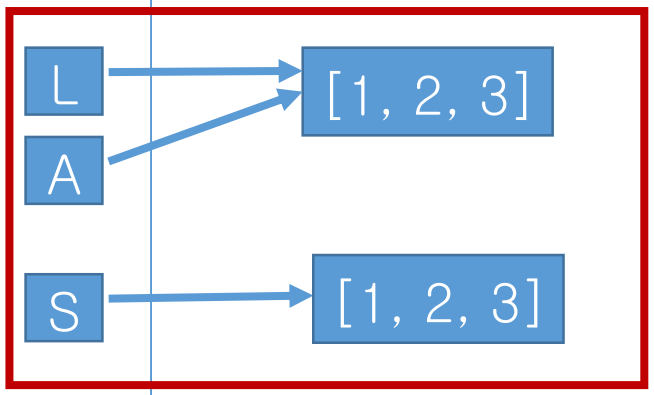
Implementation of Lists and Sets in Python:

Alias, Shallow Copy, Deep Copy

Aliasing, Shallow Copy and Deep Copy in 1D List [1/4]

- Consider the following code

```
1  import copy
2  L = [1, 2, 3]
3
4  # A is an alias of L, and S is a copy of L
5  A = L
6  S = copy.copy(L)
7  print("L is: ", L)
8  print("A is: ", A)
9  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.

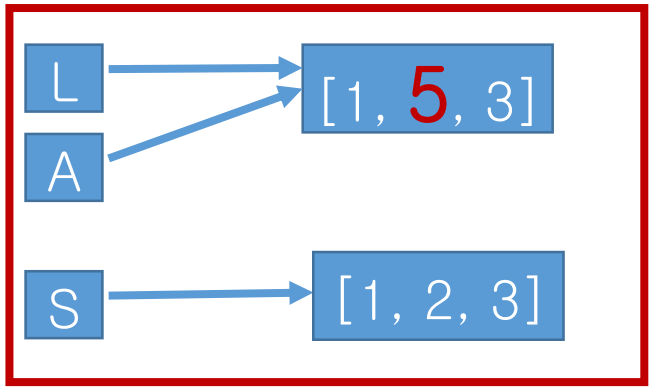
Aliasing, Shallow Copy and Deep Copy in 1D List [2/4]

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

```
12  L[1] = 5
13  print("############")
14  print("L is ", L)
15  print("A is ", A)
16  print("S is ", S)
17
```

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).

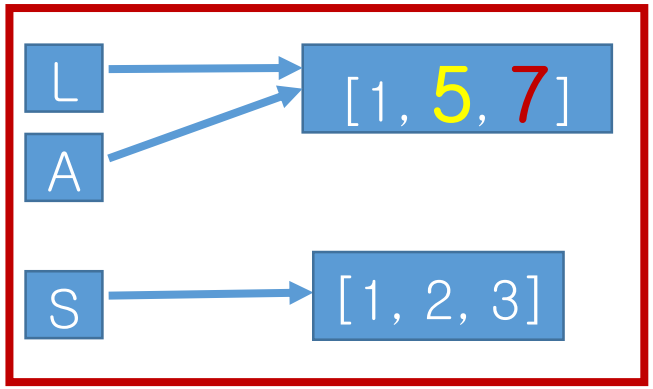
Aliasing, Shallow Copy and Deep Copy in 1D List [3/4]

- Also, if we change the 3rd element (i.e. index 2) of A (not L)

```
19  A[2] = 7
20  print("############")
21  print("L is ", L)
22  print("A is ", A)
23  print("S is ", S)
24
25
```

REPL [python]

```
############
L is  [1, 5, 7]
A is  [1, 5, 7]
S is  [1, 2, 3]
```



Again, 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 [4/4]

- 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)

```
1  import copy
2
3  a = [1, 5, 9]
4  b = copy.copy(a)
5  c = copy.deepcopy(a)
6  d = list(a)
7  e = a + [ ]
8  f = a[:]
9
10 # change third element (index 2) of a
11 a[2] = 40
12 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!

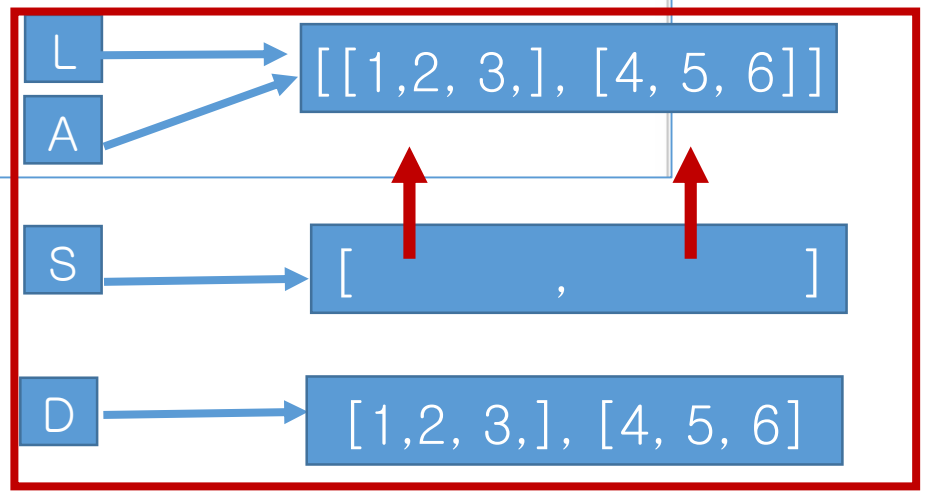
Aliasing, Shallow Copy and Deep Copy in 2D List [1/7]

- Consider the following code

```
1  import copy
2
3  L = [[1 ,2 ,3], [4, 5, 6]]
4
5  # A is an alias of L, S is a shallow copy of L, and D is a deepcopy of L
6  A = L
7  S = copy.copy(L)
8  D = copy.deepcopy(L)
9
10 print("L is: ", L)
11 print("A is: ", A)
12 print("S is: ", S)
13 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 [2/7]

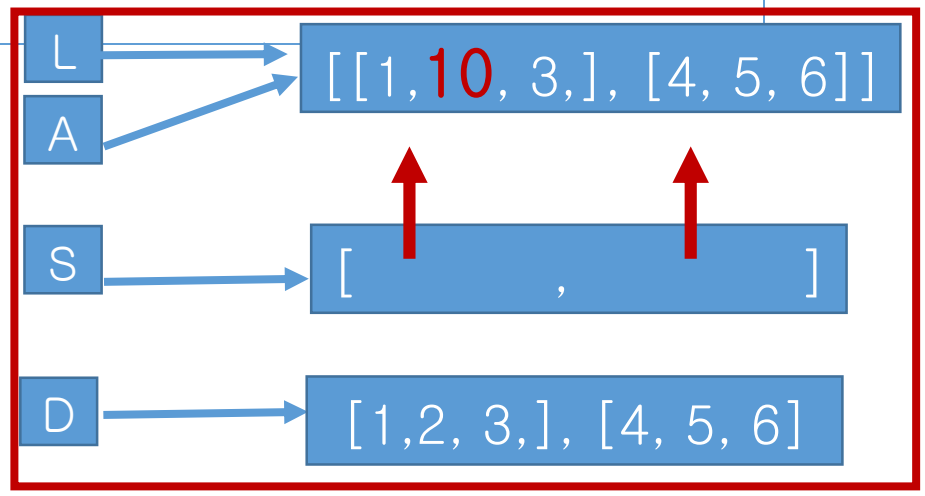
- If we change the 2nd element (i.e. index 1) of the first list in L

```
 9  # Change the 2nd element (index 1) of the first list in L to 10
10  L[0][1] = 10
11  print("L is: ", L)
12  print("A is: ", A)
13  print("S is: ", S)
14  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.



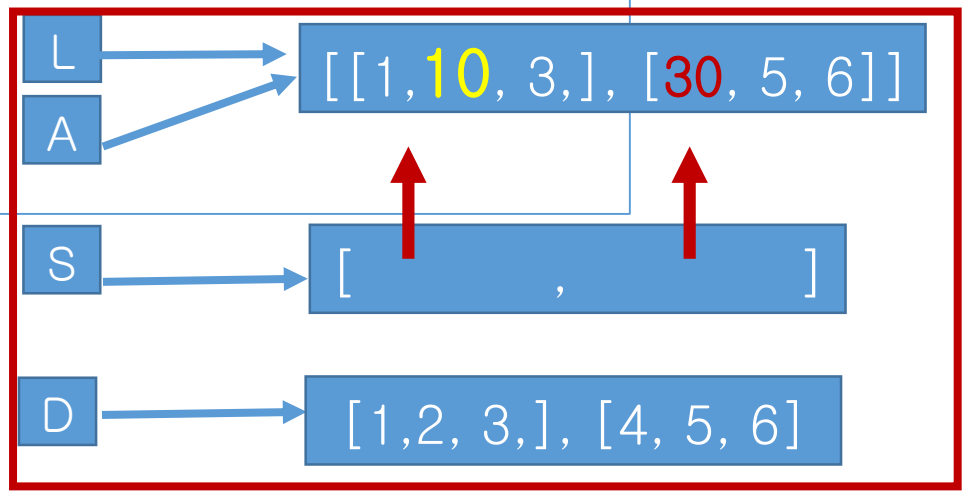
Aliasing, Shallow Copy and Deep Copy in 2D List [3/7]

- If we change the 1st element (i.e. index 0) of the second list in A (not L)

```
24  # Change the 1st element of the second list in A to 30
25  A[1][0] = 30
26  print("L is: ", L)
27  print("A is: ", A)
28  print("S is: ", S)
29  print("D is: ", D)
30
```

REPL [python]

```
L is:  [[1, 10, 3], [30, 5, 6]]
A is:  [[1, 10, 3], [30, 5, 6]]
S is:  [[1, 10, 3], [30, 5, 6]]
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.

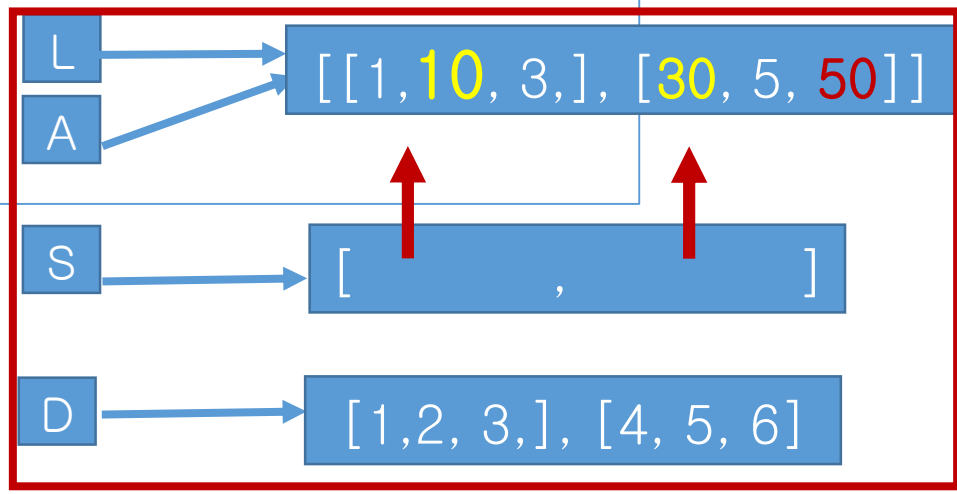
Aliasing, Shallow Copy and Deep Copy in 2D List [4/7]

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

```
32  # Change the 3rd element of the second list in S to 50
33  S[1][2] = 50
34  print("L is: ", L)
35  print("A is: ", A)
36  print("S is: ", S)
37  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.

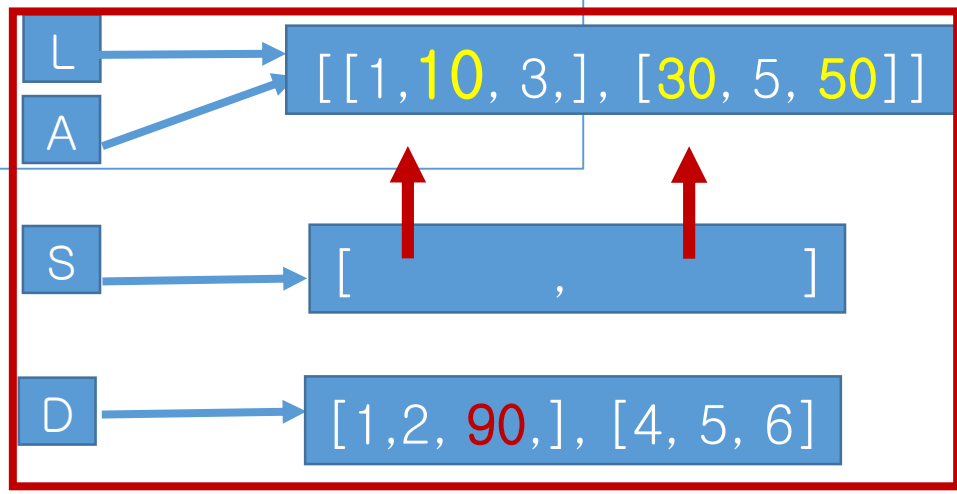
Aliasing, Shallow Copy and Deep Copy in 2D List [5/7]

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

```
40  # Change the 3rd element of the first list in D to 90
41  D[0][2] = 90
42  print("L is: ", L)
43  print("A is: ", A)
44  print("S is: ", S)
45  print("D is: ", D)
46
```

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.

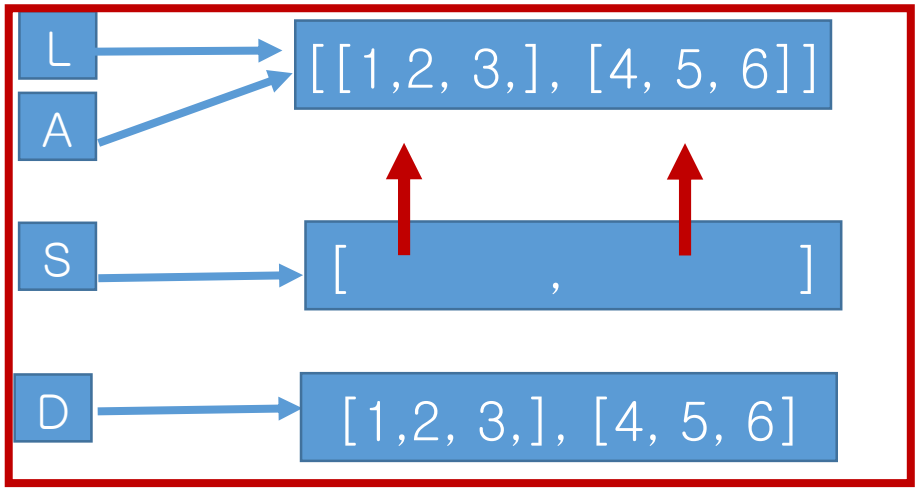
Aliasing, Shallow Copy and Deep Copy in 2D List [6/7]

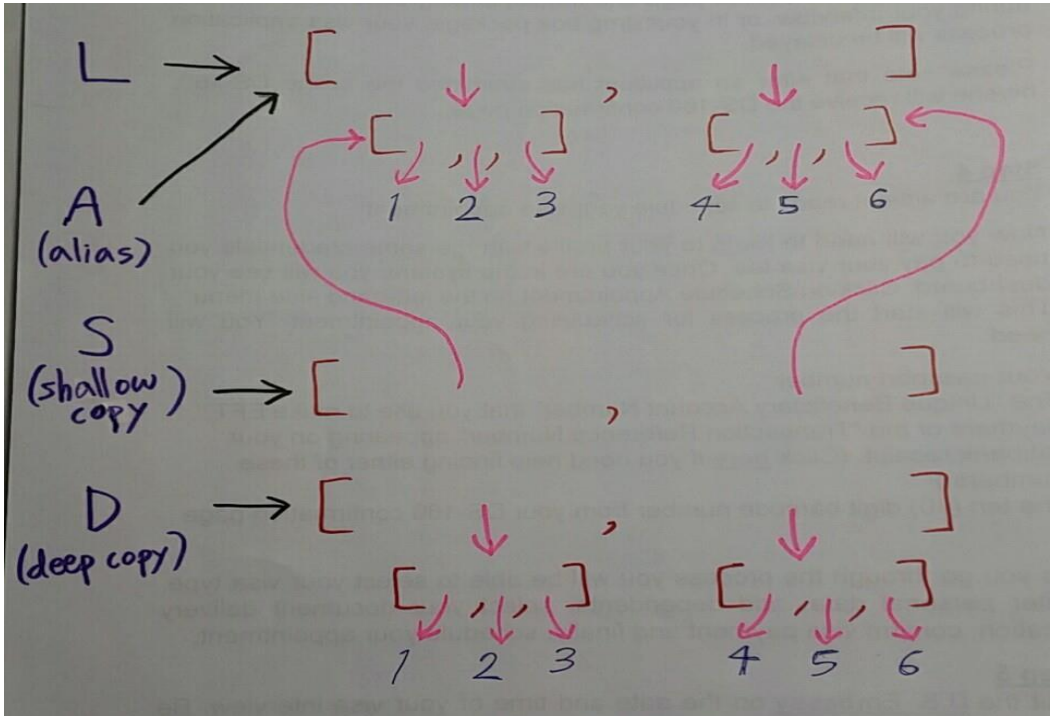
- What really happens in Python when we write this code is…

```
1  import copy
2
3  L = [[1 ,2 ,3], [4, 5, 6]]
4
5  # A is an alias of L, S is a
6  A = L
7  S = copy.copy(L)
8  D = copy.deepcopy(L)
9
10 print("L is: ", L)
11 print("A is: ", A)
12 print("S is: ", S)
13 print("D is: ", D)
```

REPL [python]

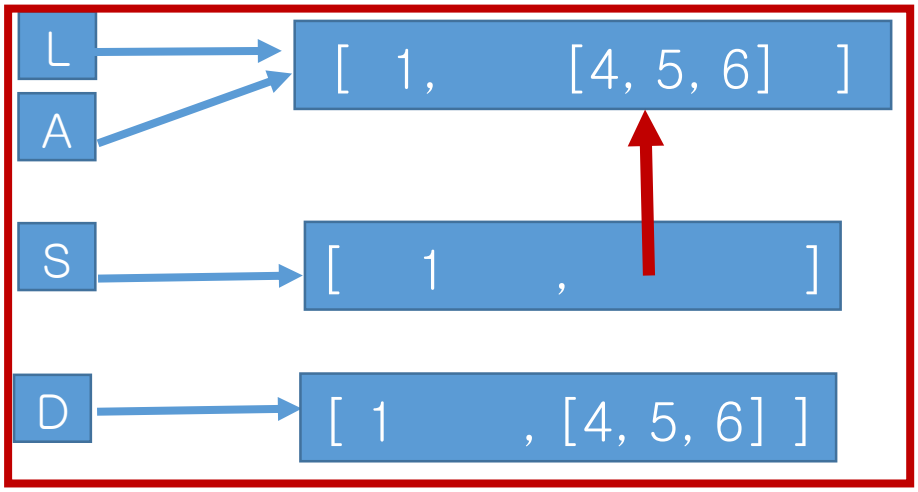
```
L is:  [[1, 2, 3], [4, 5, 6]]
A is:  [[1, 2, 3], [4, 5, 6]]
S is:  [[1, 2, 3], [4, 5, 6]]
D is:  [[1, 2, 3], [4, 5, 6]]
>>>
```





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
>>> L, A, S, D
```

- Suppose

```
>>> S[0] = 9
>>> L, A, S, D
```

Aliasing, Shallow Copy and Deep Copy in 2D List [7/7]

In summary, for 2D lists:

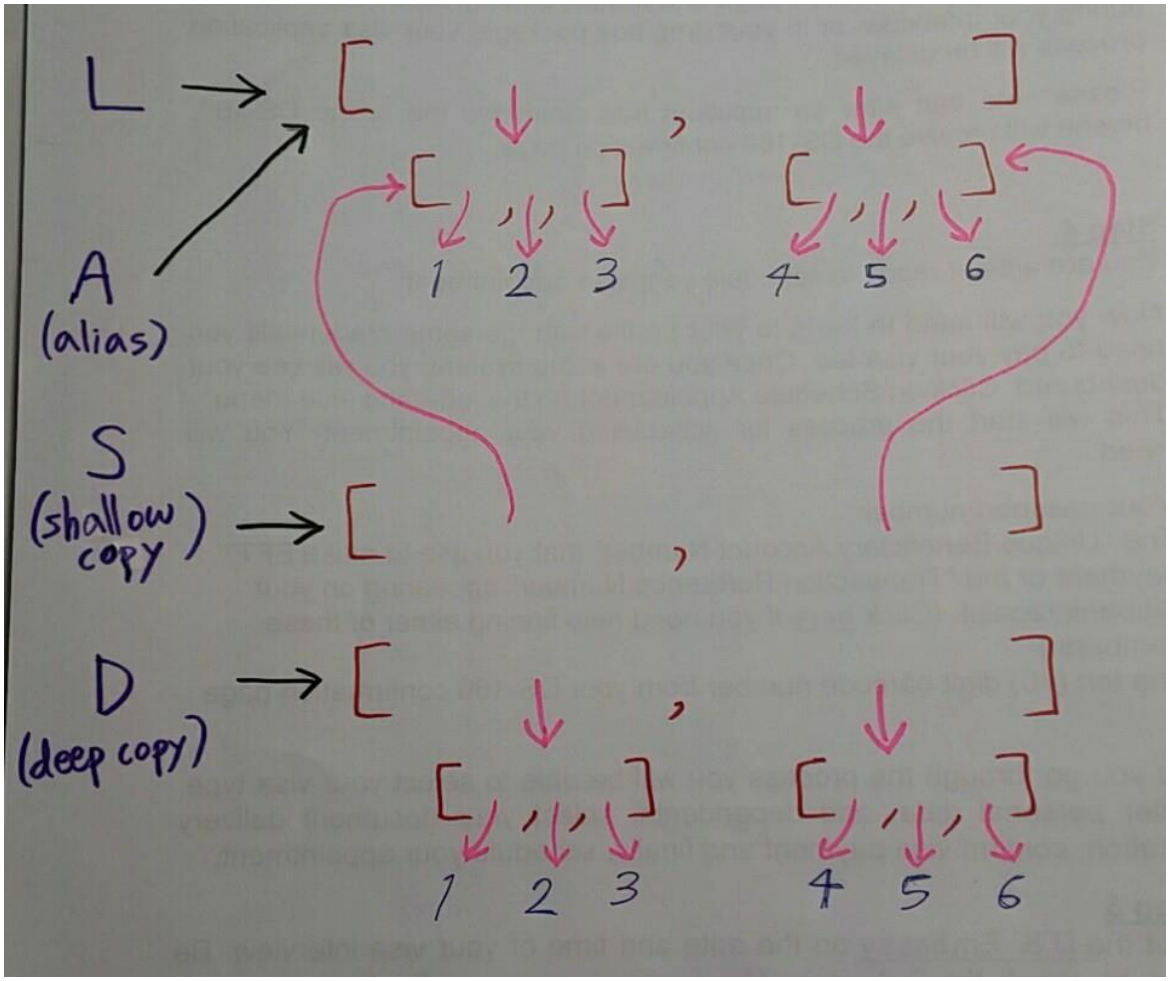
- --***Aliases*** are lists that point to the exact same 2D list (therefore any change in the original 2D list is applied to the alias too)

- --***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)
  - (e.g. 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)

Bonus Slides:
Why Implement Lists This Way? (As pointers)

```
1  import copy
2
3  L = [[1 ,2 ,3], [4, 5, 6]]
4
5  # A is an alias of L, S is a
6  A = L
7  S = copy.copy(L)
8  D = copy.deepcopy(L)
```

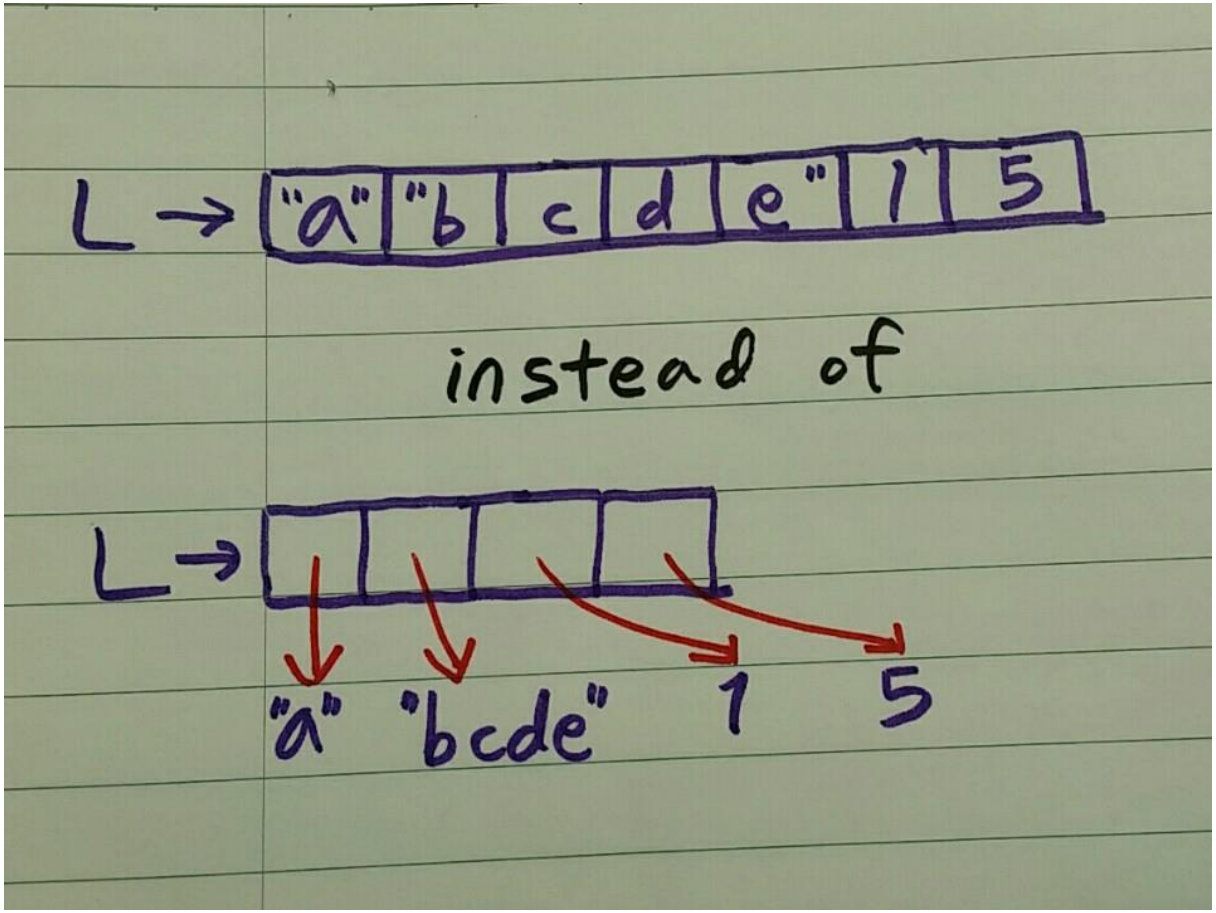


Why Implement Lists as Pointers (1/3)

Imagine a Python list stores its elements in its body itself instead of having pointers in its body and the elements are pointed by the pointers.

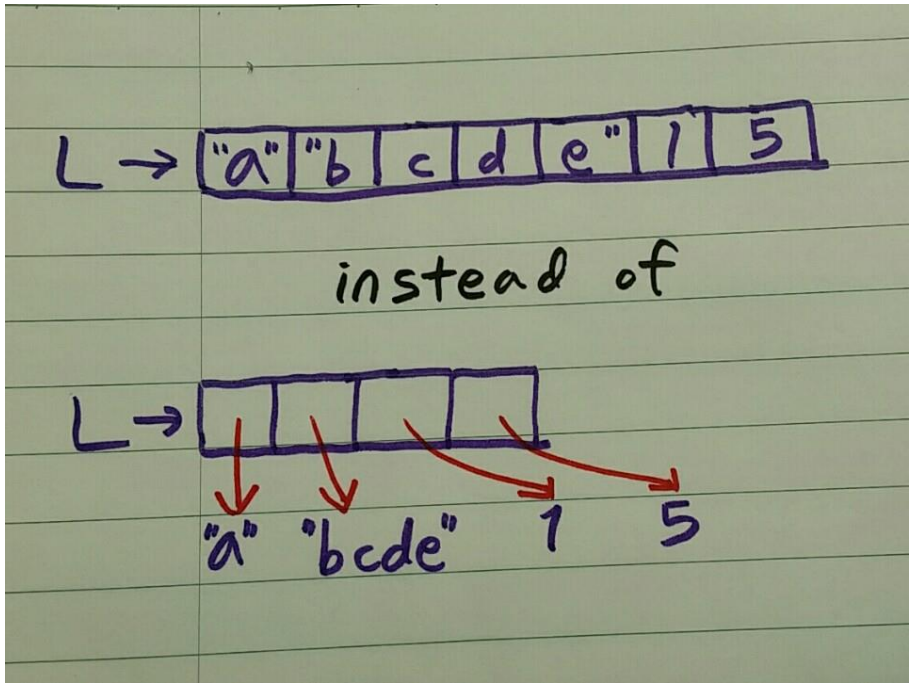
Consider a list L = [“a”, “bcde”, 1, 5].

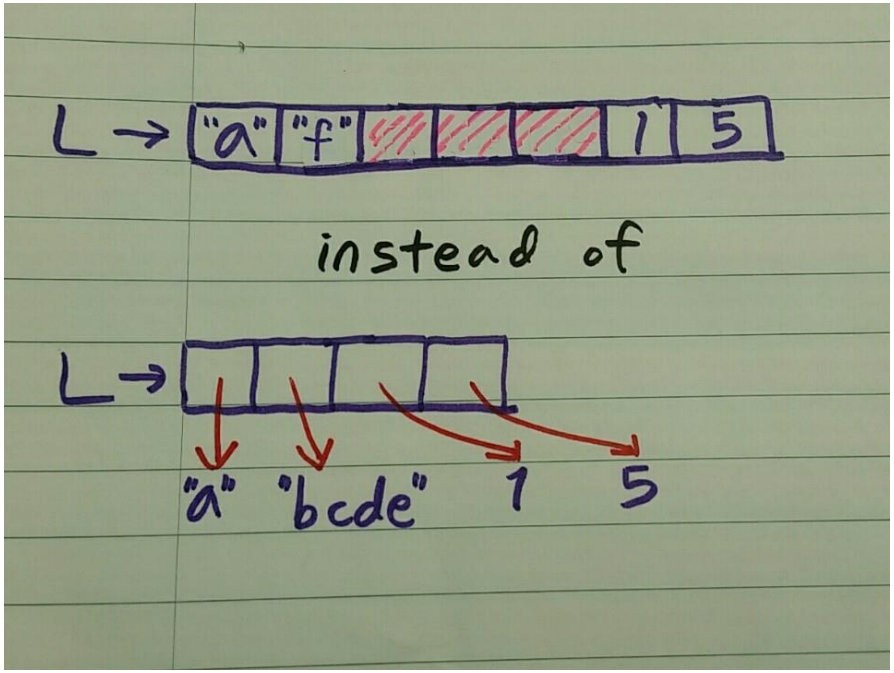
It will look like:



Why Implement Lists as Pointers (2/3)

- **Problem 1: Memory waste due to Update**
  - Consider this modification: L[1] = "f"
  - Then, the second 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!





Why Implement Lists as Pointers (3/3)

- **Problem 2: Search Inefficiency**

  - Consider this search: L[4]. i.e. We want to check what the element L[4] 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?

  - On the other hand, if lists are implemented as how they are in Python (with pointers), Python can go to the index 4^{th} (i.e. third) square, and check whatever the pointer in the third square is pointing to.
    - This implementation is what makes indexing valuable.

Sets in Python3

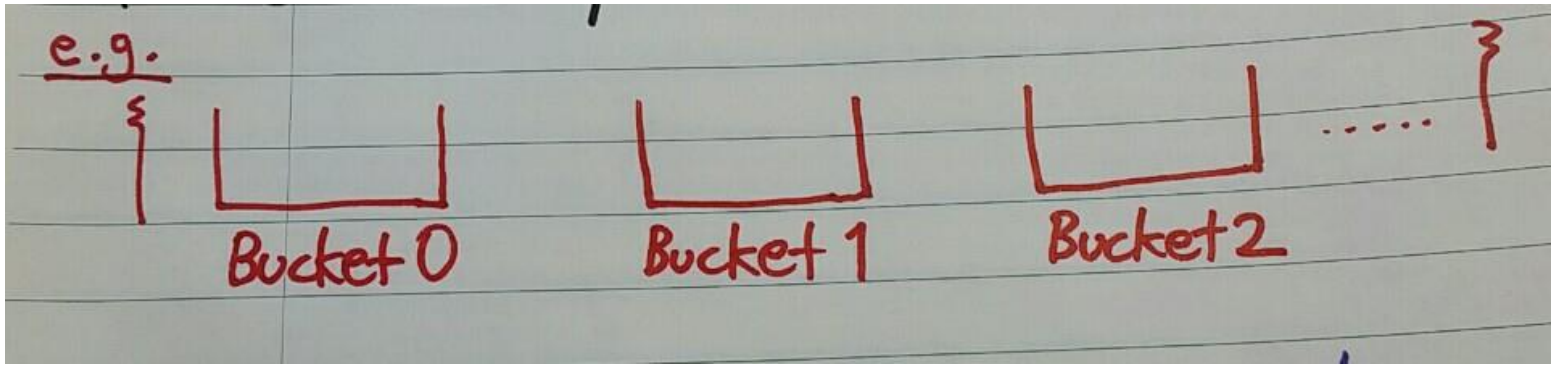
(a) How sets are created (Hash)

(b) Properties of sets

(c) Copies in sets

How Sets are Created (1/2)

- A set is composed of buckets.



- When we add an element in a set, Python uses its Hash function to decide which bucket the element will go into.
- Therefore, elements of a set should be hashable values

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 for set:
  - int, float, strings, boolean, user-defined class and their objects
  - Values of List data type are not hashable (no set like {[2], [5]})
  - Values of Set data type are not hashable (no set like {{3}, {1,7}}) [5]})

- It's Python's job to decide the number, 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 (Outline)

- Sets are unordered collections of values of same data types
- Sets' elements are unique. (No repetition)
- Sets are all 1 dimensional of same data type's guys
  - There is no set like {2, 3, {4, 6}}, {3, 5, "kim"}
- 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

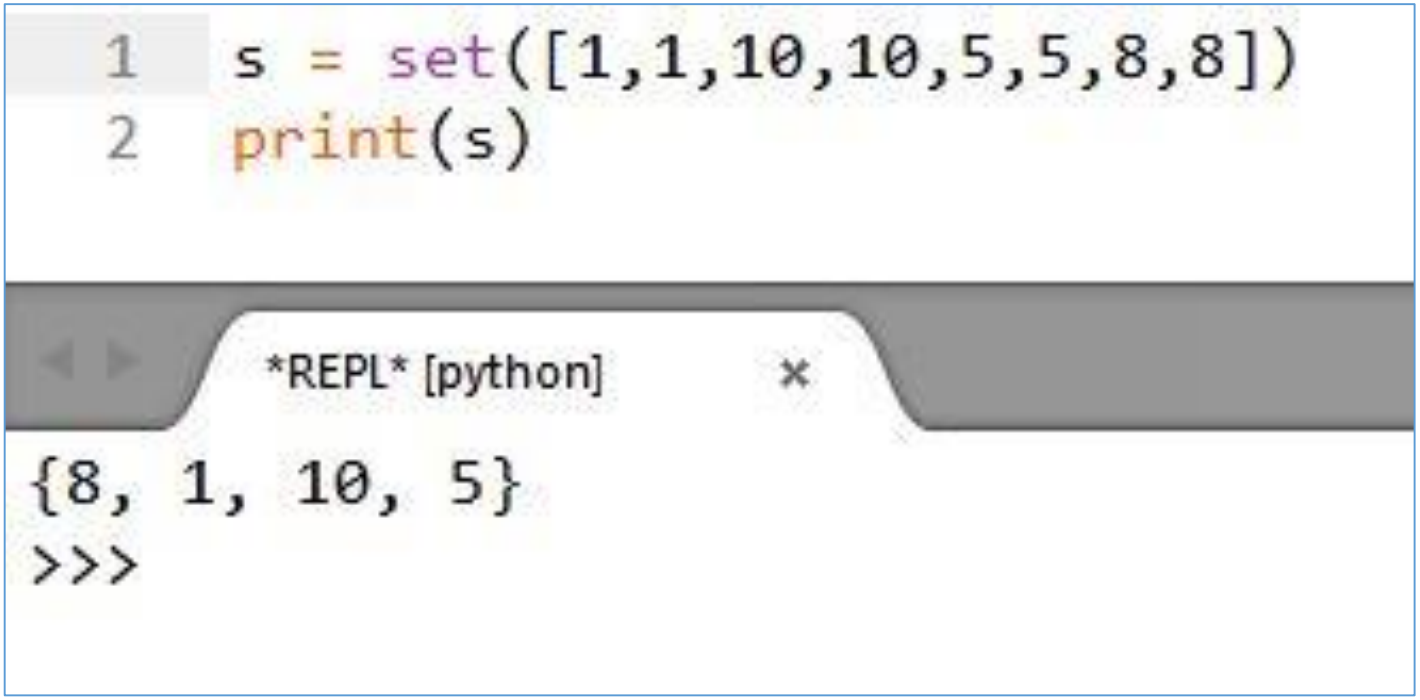
```
1  s = set([1,10,5,8])
2  print(s)
```

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



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!

- 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]

- Suppose we want to check whether or not a particular element x is in our set s.
- x in s returns True if x is an element of s and False if not.

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

REPL [python]

```
True
False
>>>
```

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

- Same for lists! For a list L, x in L returns True if x is an element of 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 if 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

- To check that sets are much more efficient than lists, let's code it!

- Create a list containing 2, 4, 6, … , 29998, 30000, and 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 time the two cases (for a list and for a set).

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

```
1  import time
2
3  L = []
4  s = set()
5  for n in range(2, 30001, 2):
6      # even numbers between 2 and 30000 (inclusive)
7      L.append(n)
8      s.add(n)
9  # Now, L is a list containing 2, 4, 6, ... , 29998, 30000
10 # Now, s is a set containing 2, 4, 6, ... , 29998, 30000
11
12 ############################ LIST ################################
13 start = time.time()
14 count = 0
15 for x in range(30001):
16     if x in L:
17         count += 1
18 end = time.time()
19 listTime = end - start
20 print("For a list, count =", count," and time = %0.6f seconds" % listTime)
21 ########################## SET ###################################
22 start = time.time()
23 count = 0
24 for x in range(30001):
25     if x in s:
26         count += 1
27 end = time.time()
28 setTime = end - start
29 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!

Bonus Slide- Dictionaries

- Dictionaries are similar to sets…
- except that in dictionaries, each element is a pair of a key and a value.
- (The pair is in 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)

Aliasing and Copy of Sets [1/4]

- We can copy sets in Python.
- Just like 1D lists, when we add/remove elements in a set (remember, we cannot modify the elements which are immutable), that change is applied to aliases of that set but not to the copies.

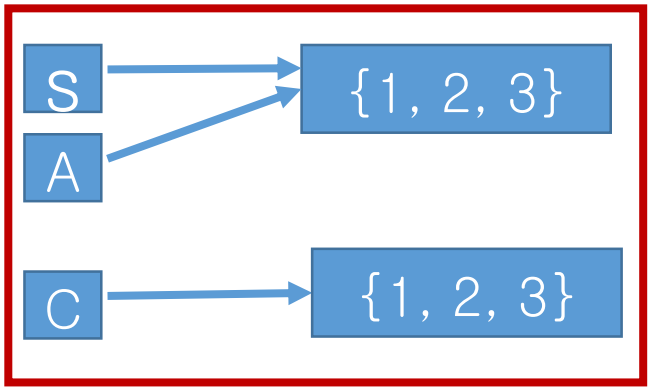
```
1  import copy
2  s = {1,2,3}
3  # a is an alias, and c is a copy of s.
4  a = s
5  c = copy.copy(s)
6  print("Original s is ", s)
7  print("Alias a is ", a)
8  print("Copy c is ", c)
```

REPL [python]

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

Set에서는 aliasing과
copy.copy만 차이가 있고

copy.copy와
copy.deepcopy가 차이가 없다



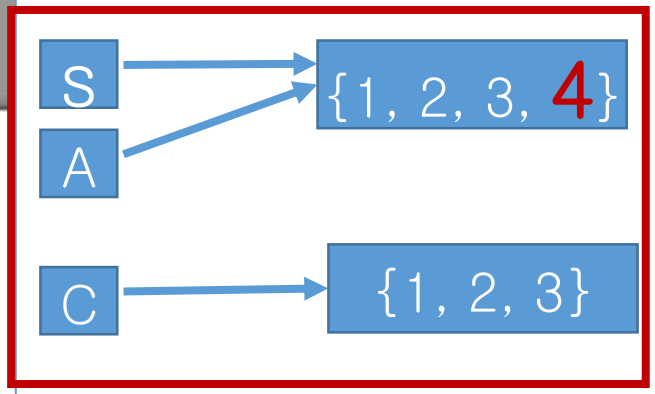
Alaising and Copy of Sets [2/4]

- If we add 4 in the original set s…

```
11  s.add(4)
12  print("Original s is ", s)
13  print("Alias a is ", a)
14  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}
>>>
```



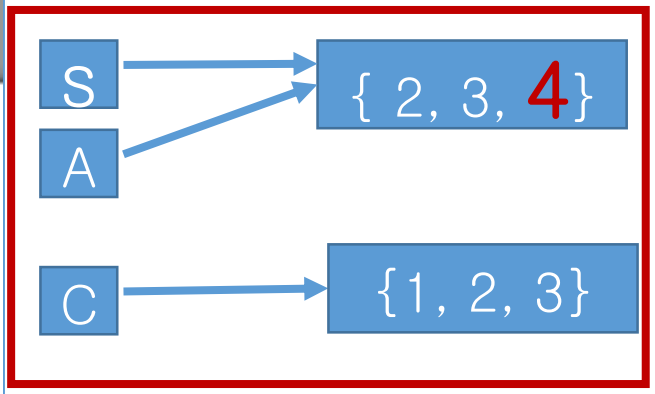
s and the alias a are affected, but the copy c is not affected.

Aliasing and Copy of Sets [3/4]

- If we remove 1 from the alias a…

```
16  a.remove(1)
17  print("Original s is ", s)
18  print("Alias a is ", a)
19  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.

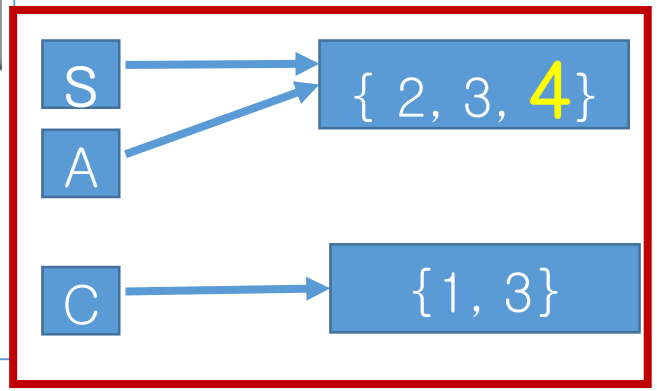
Alaising and Copy of Sets [4/4]

If we remove 2 from the copy c…

```
21  c.remove(2)
22  print("Original s is ", s)
23  print("Alias a is ", a)
24  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.