Lists and Sets in Python3:

Alias, Shallow Copy, Deep Copy

1-d Lists [1/4]

- 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.

1-d Lists [2/4]

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

```
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).

1-d Lists [3/4]

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

```python
A[2] = 7
print("############")
print("L is ", L)
print("A is ", A)
print("S is ", S)
```

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).

1-d Lists [4/4]

- When a 1-d 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 1-d 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]
>>>
```

2-d Lists [1/7]

- Consider the following code

```python
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.

2-d Lists [2/7]

- (code continued) 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.

2-d Lists [3/7]

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

```python
# Change the 1st element of the second list in A to 30
A[1][0] = 30
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, 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.

2-d Lists [4/7]

- (code continued) 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.

2-d Lists [5/7]

- (code continued) 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.

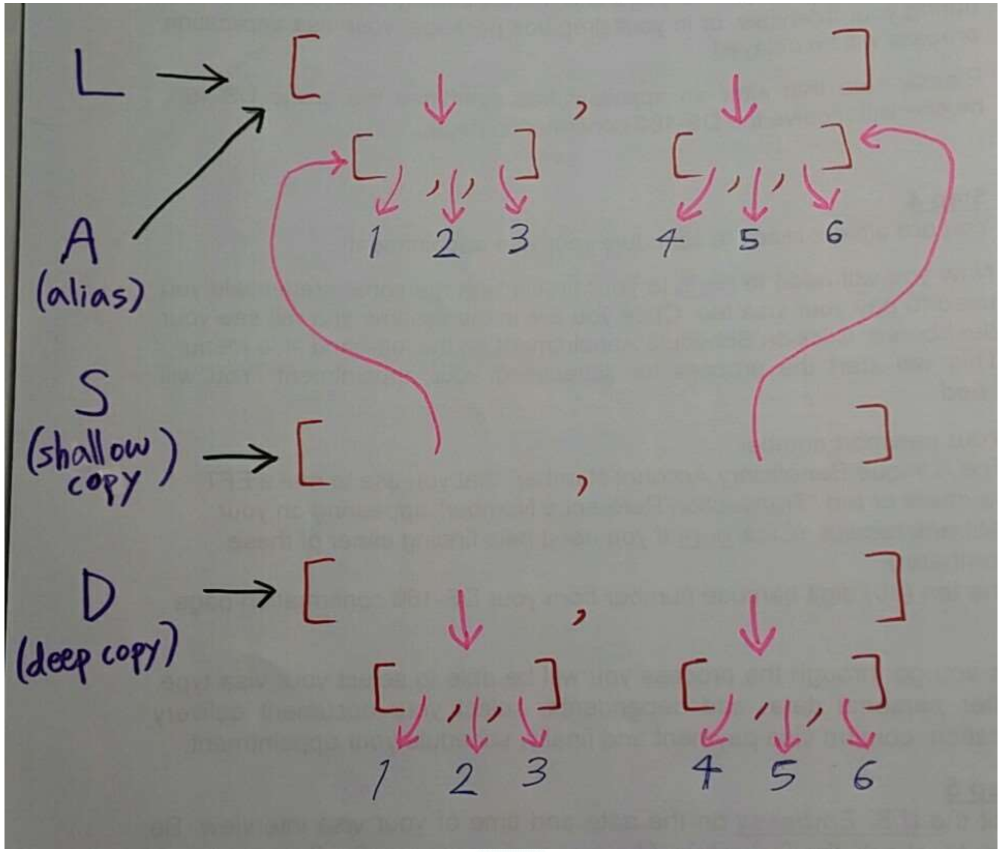
2-d Lists [6/7]

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

```python
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]]
>>>
```

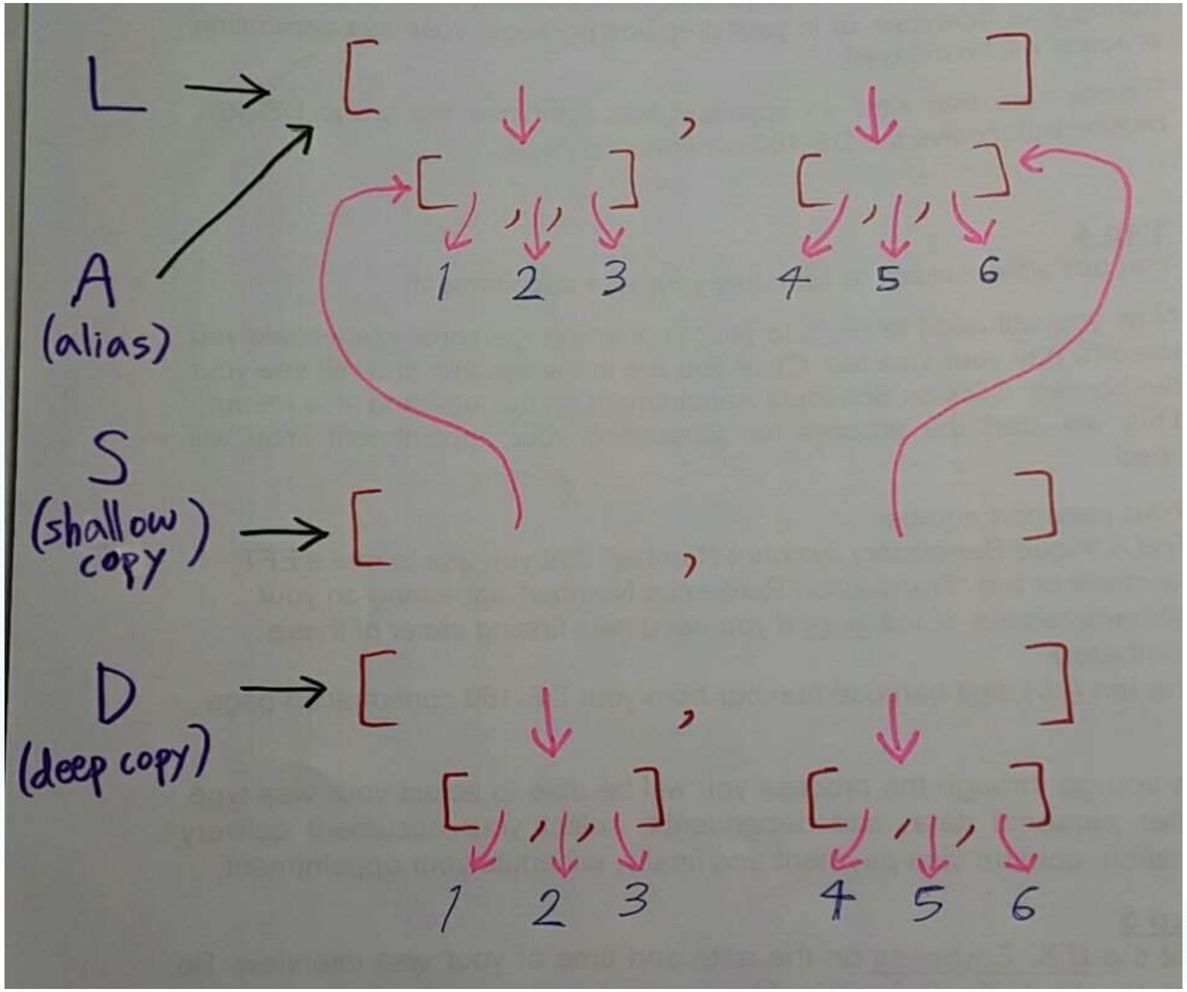


2-d Lists [7/7]

In summary, for 2-d lists:

- --***Aliases*** are lists that point to the exact same 2-d list (therefore any change in the original 2-d 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 2-d lists (therefore change in elements of those inner 1-d lists in the original 2-d list affect the shallow copy too)
- (But adding/removing/changing elements (which can be 1-d lists) themselves in the original 2-d list does not affect the shallow copy)
- (e.g. If we add a new 1-d 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 2-d list do not affect the deep copy at all)

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

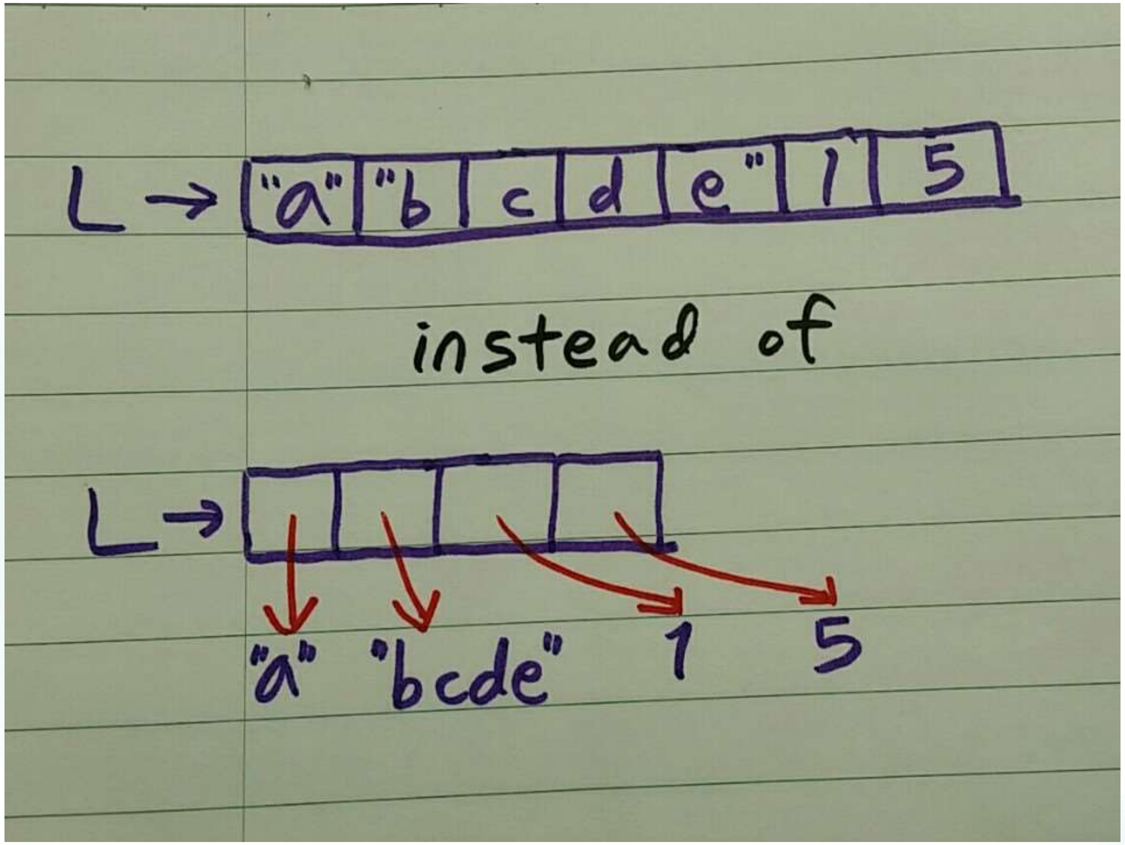


Why Implement Lists as Pointers (1/3)

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.

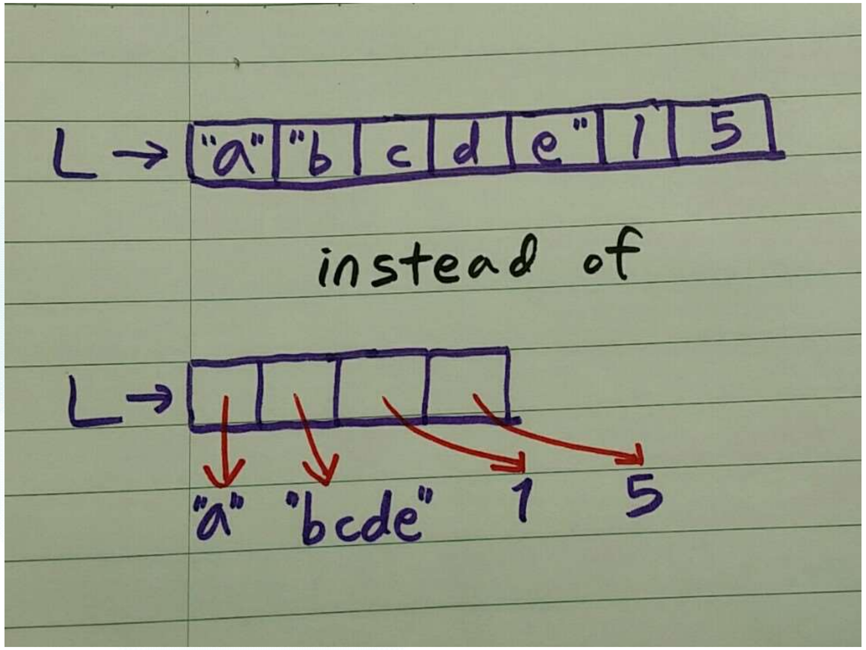
Consider a list L = ["a", "bcde", 1, 5].

It will look like:

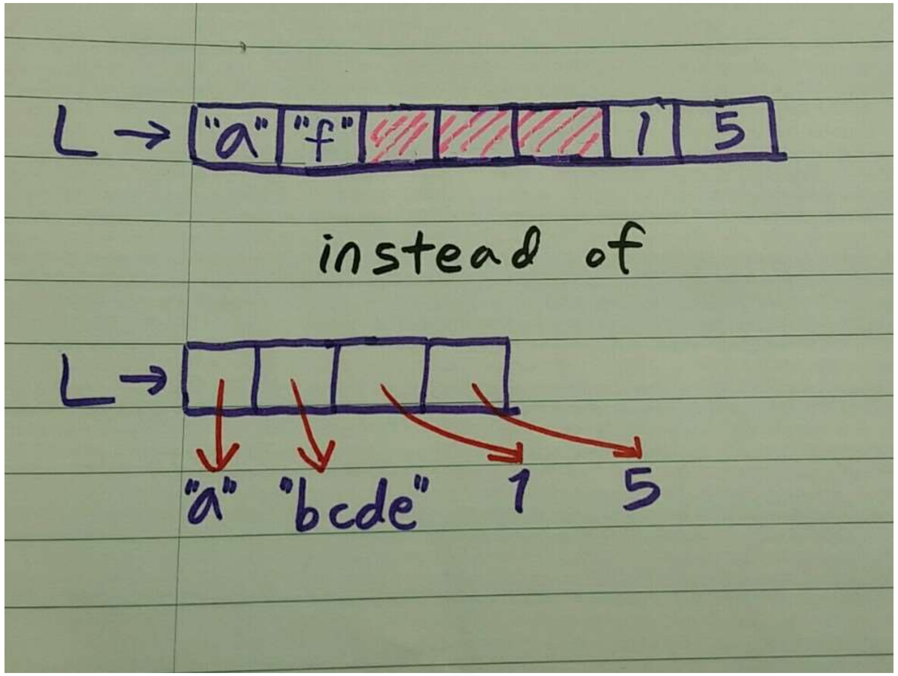


Why Implement Lists as Pointers (2/3)

- **Problem 1 of this implementation:**
- 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 of this implementation:**
- 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

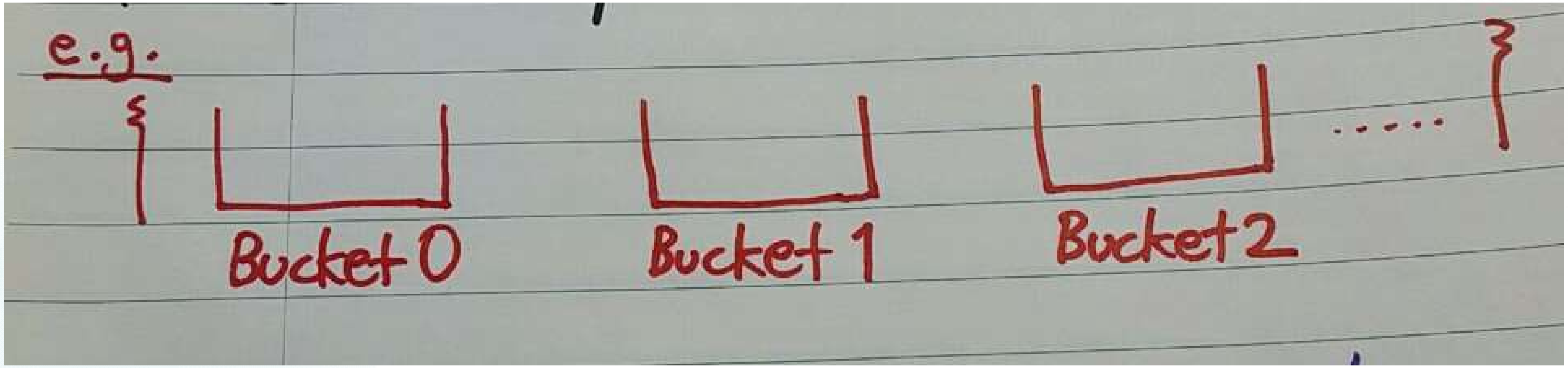
(A) HOW SETS ARE CREATED (HASH)

(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.

How Sets are Created (2/2)

- Suppose there are n-many buckets in our set.

- For an arbitrary element x (Any hashable type in Python: ints, floats, strings, booleans, user-defined classes and their objects, etc), hash(x) returns some number.
- It's Python's job to decide the number, so don't worry about how to get such a number.
- *(Note that lists, sets, and dicts are not hashable)*

- 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 #y.

Properties of Sets (Outline)

- Sets are unordered.
- 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!!)

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, and then 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.

-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.

```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)

-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)

```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!

Bonus Slide- Dictionaries

- Dictionaries are similar to sets…
- except that in dictionaries, each element is a <u>pair</u> 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)

Copies of Sets (1/4)

- We can copy sets in Python.
- Just like 1-d 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.

- Let's try it out in Python!

```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}
>>>
```

Copies of Sets (2/4)

- (code continued) 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.

Copies of Sets (3/4)

- (code continued) If we remove 1 from the alias a…

```
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.

Copies of Sets (4/4)

- (code continued) 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.