MC3203: PPS
#21: Memory Layout for Lists/Objects

Spring 2019

Copy/Equality Operations for Built-In Data Structures: Summary

Copy operations for lists

- `=` : alias
- `[:]` : shallow copy (copy of only top-level list)
  - deep copy for "one-dimensional" lists (e.g. `[1,2,3]`)
  - not deep copy for list of lists (of lists (of lists ..))
- `copy.deepcopy(·)` : deep copy
  - recursively copies lists of all levels

Equality operations for lists

- `is` : check if aliased
- `==` : deep equality (`a is b` $\Rightarrow$ `a == b` , but not vice versa)

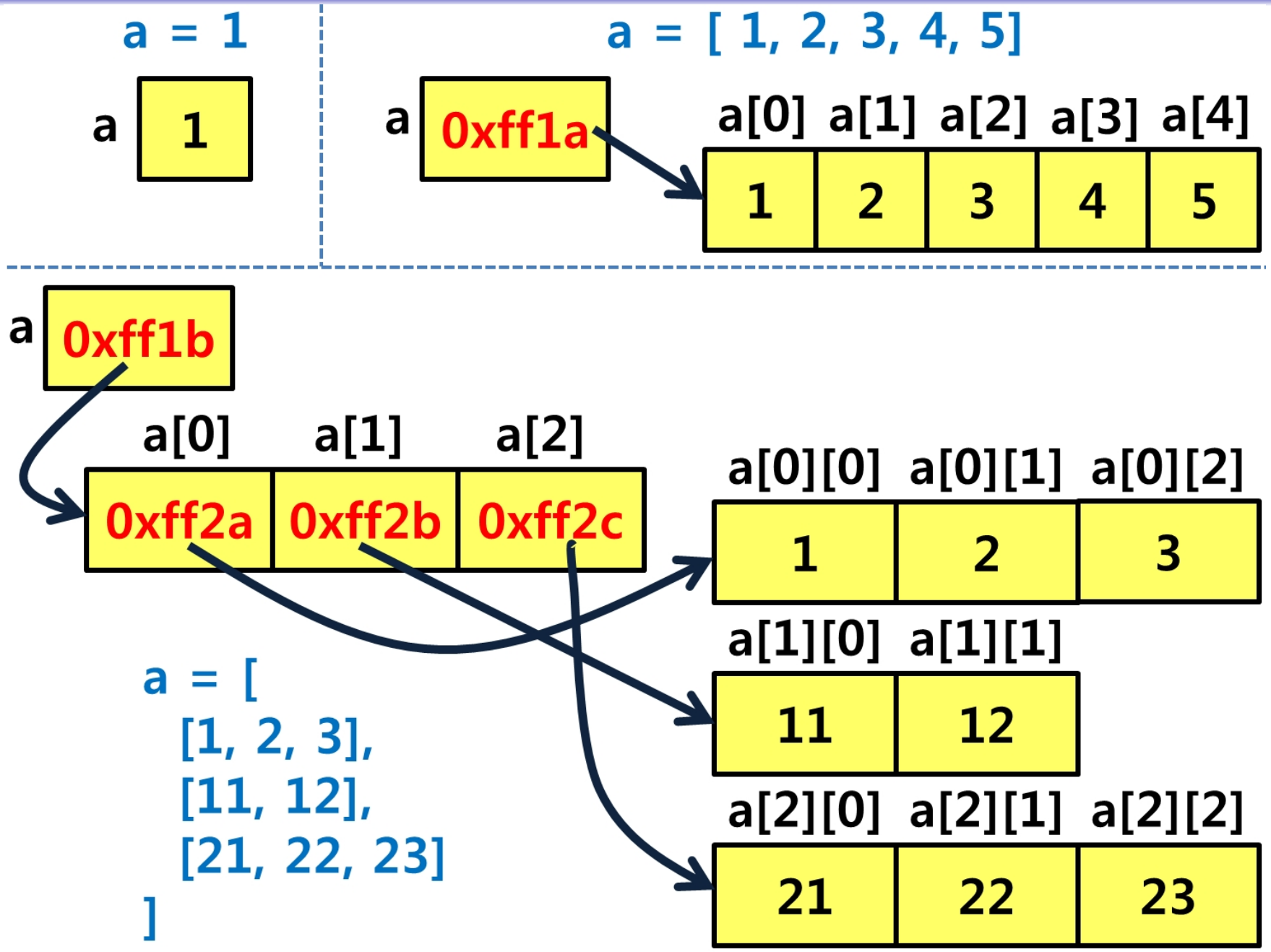
Copy/Equality operations for user-defined objects:

- `=` / `copy.deepcopy(·)` / `is` : the same as for lists
- `==` : $\begin{cases} \text{as you wish (e.g. deep equality)} & \text{if overloaded by } \texttt{\_\_eq\_\_} \\ \texttt{is (check if aliased)} & \text{otherwise} \end{cases}$

Copy/Equality operations for sets/dictionaries: the same as for lists

Outline

Memory Layout for Lists (1/3) (simplified for easier understanding)

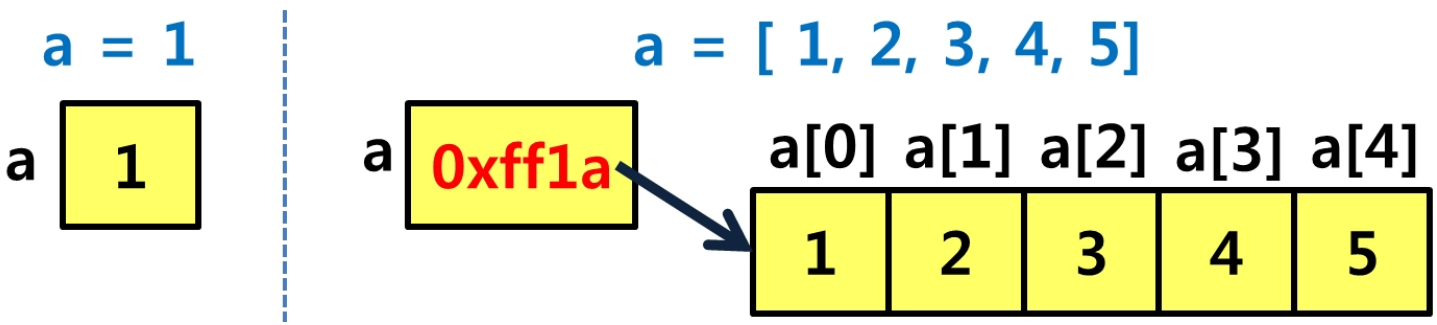


Memory Layout for Lists (2/3) (simplified for easier understanding)

What kind of value does a variable have?

- For a primitive type variable: number/string/.. (e.g. `1`)
  - primitive type: `int`, `float`, `bool`, ...
- For a list type variable: **memory address** (e.g. `0xff1a`)
  - address of the start (e.g. `a[0]`) of contiguous memory block
  - since each `a[i]` is a primitive type variable, `a[i]` has number as its value

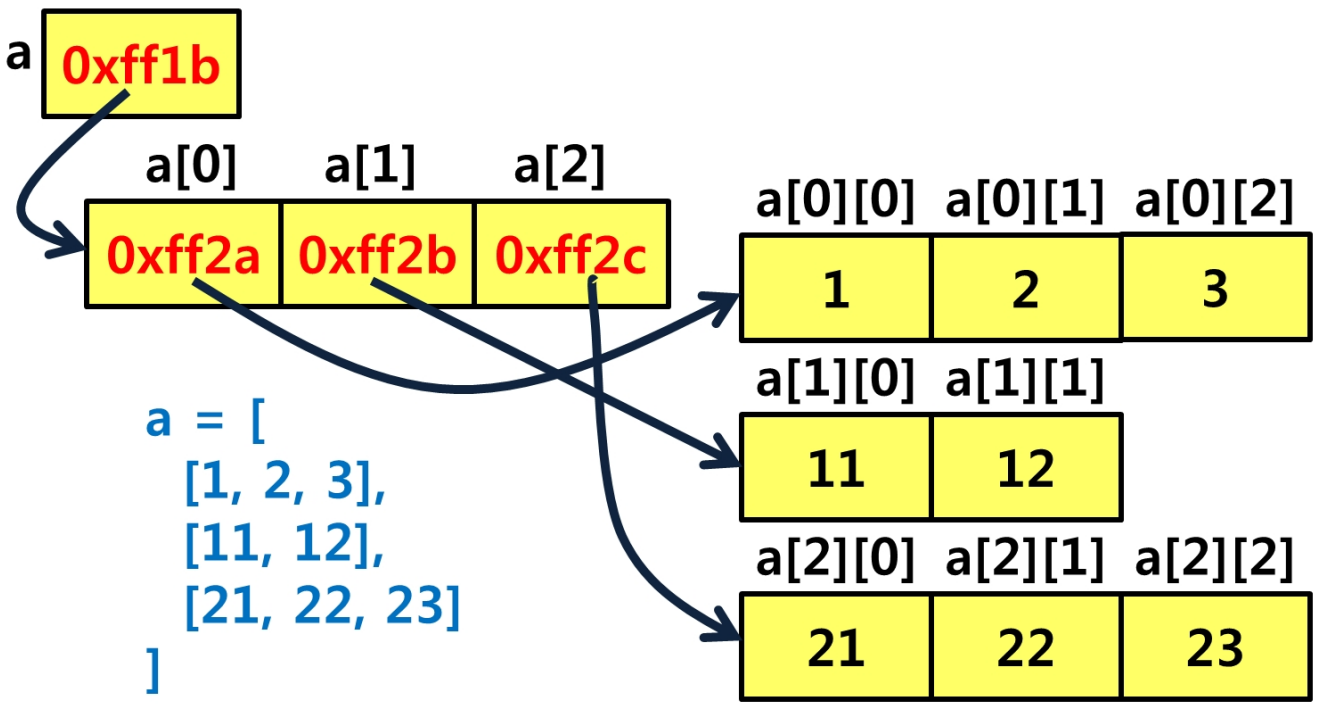
- The value of a list type variable $\neq$ list
  $=$ memory address to list



Memory Layout for Lists (3/3) (simplified for easier understanding)

What about a list of lists?

- Similarly, the value of a is the memory address to `a[0..2]`
- Also, the value of each `a[i]` is the memory address to `a[i][0]`, `a[i][1]`, $\cdots$
- Each `a[i][j]` is now a primitive type variable, so `a[i]` has number as its value

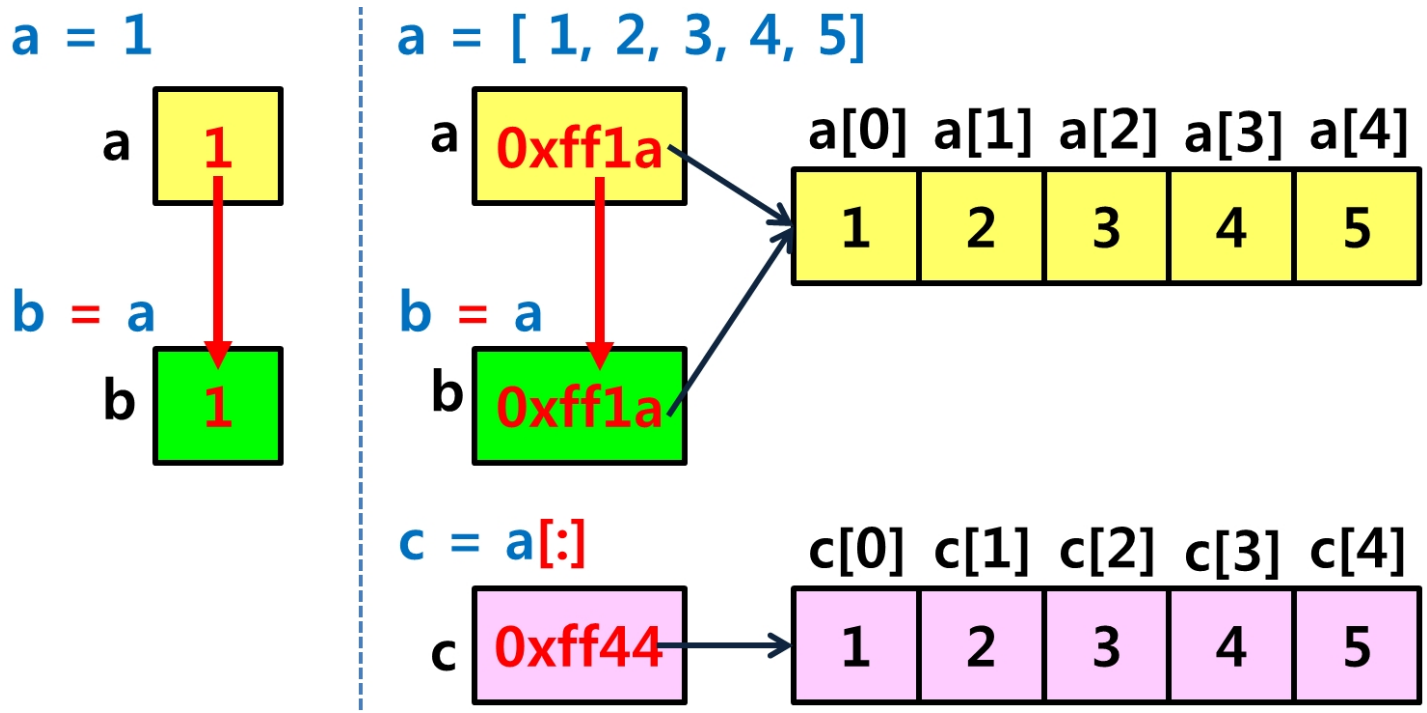


Outline

= vs. [:] for Lists

The assignment operation `=` just copies the value of a variable

- Recall: the value of a list type variable is NOT list but address
- Thus, for list a, the effect of "b = a" is to make b and a point to the **same list**
  - b is called an alias of a

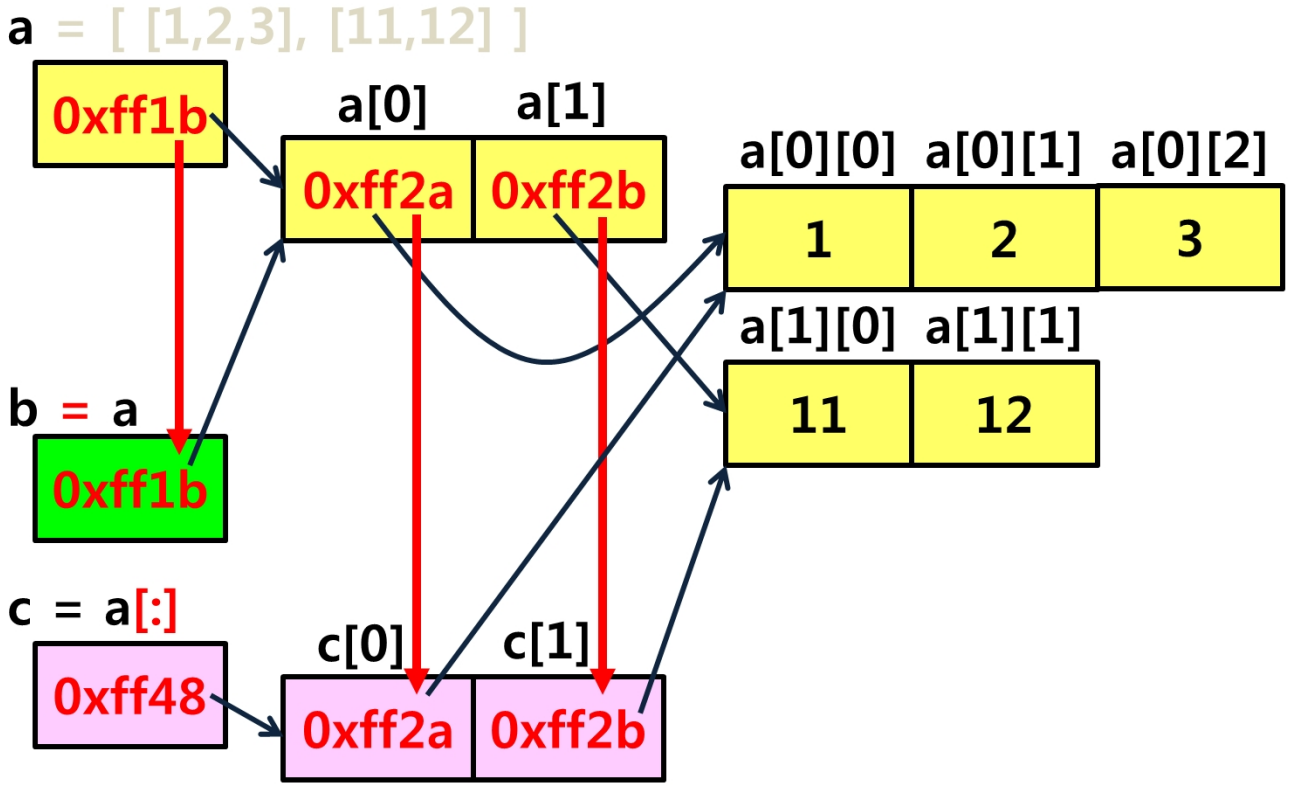


The slicing op `[ : ]` creates a new **copy of list**

= vs. [:] for Lists of Lists

= just copies the value of a variable (alias)

- a and b point to the **same list** a[0],a[1],···



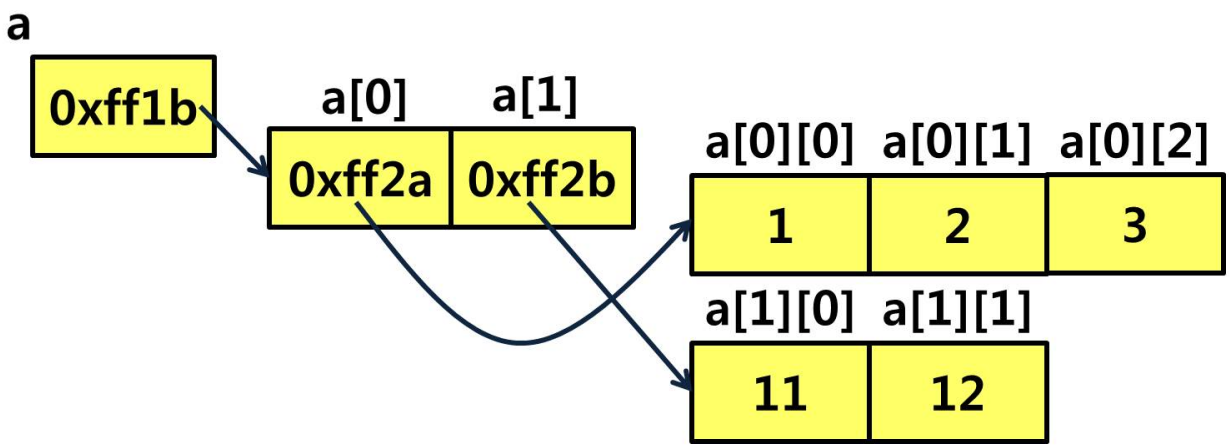
[:] creates a new copy of only **top-level** list (shallow copy)

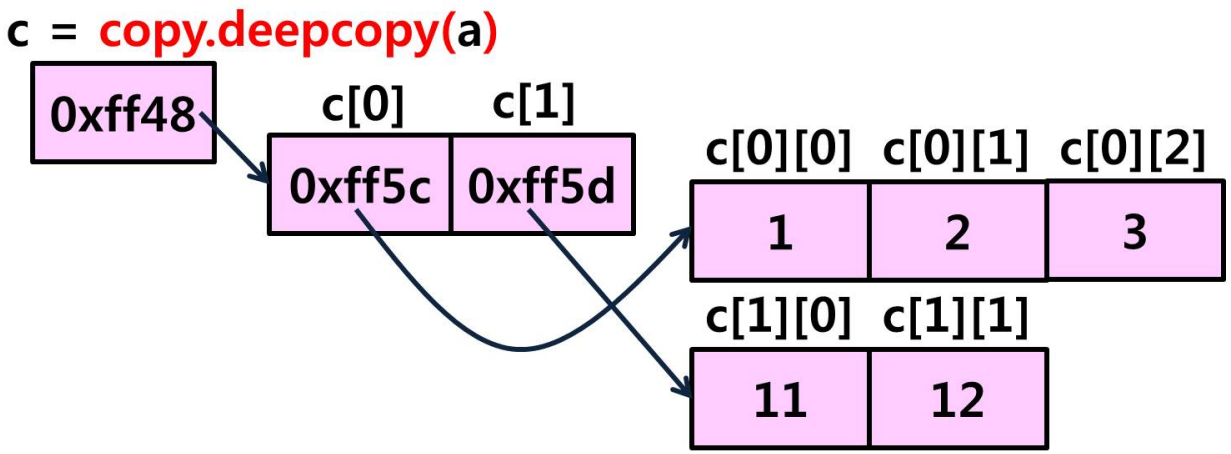
- a and c point to **different lists** (top-level deep copy)
- BUT, a[i] and c[i] point to the **same list** a[i][0],···

`copy.deepcopy(·)` for Lists of Lists

`copy.deepcopy()` create a new copy of **whole** list, **recursively**

- `c = copy.deepcopy(a)` makes `a` and `c` share nothing
- If what you want is a new deep copy of list of lists, then use `copy.deepcopy` instead of `[:]`

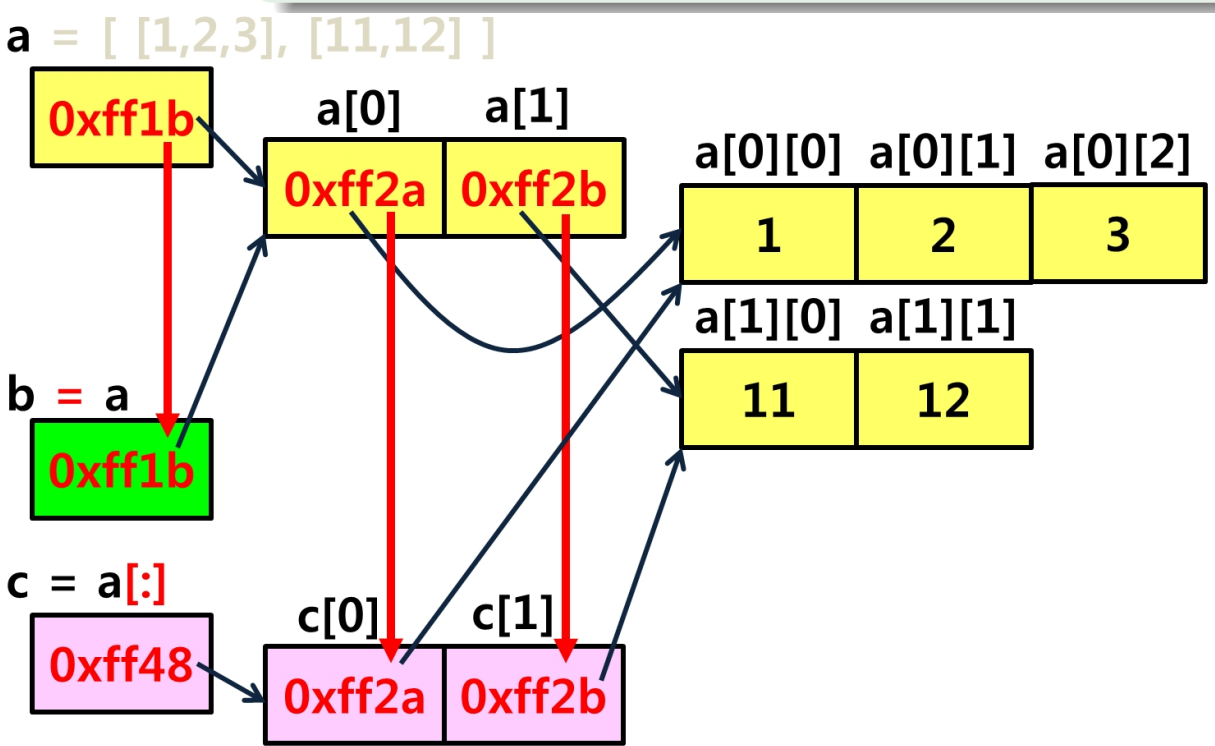




Exercise: Figure out why the output is..

```
a = [ [1, 2, 3],
      [11,12] ]

b = a
c = a[:]
d = copy.deepcopy(a)
```

```
a[1][0] = 0
a[0] = [4,5]
print(b[1][0], c[1][0], d[1][0])  # 0 0 11
print(b[0],c[0],d[0]) # [4,5] [1,2,3] [1,2,3]

a[0],a[1] = a[1],a[0]  # swapping rows
print(a,b)    # [[0,12], [4,5]] [[0,12], [4,5]]
print(c)      # [[1,2,3], [0,12]]
```



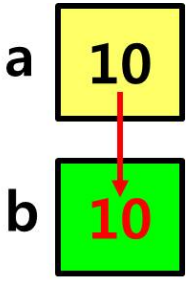
Outline

Lists as Function Input (1/2)

- Passing an argument a to function as a parameter b has the effect of b = a (i.e. aliasing)
- For each case, figure out why the output is :

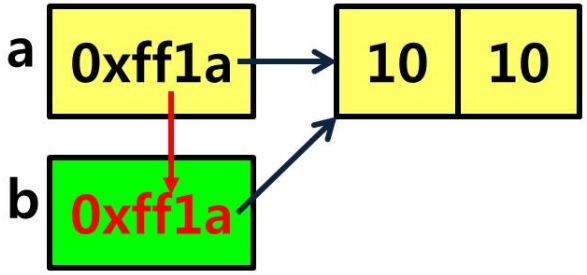
Number as input

```
def f(b): # b = a
    b = 0


a = 10
f(a)
print(a)
```



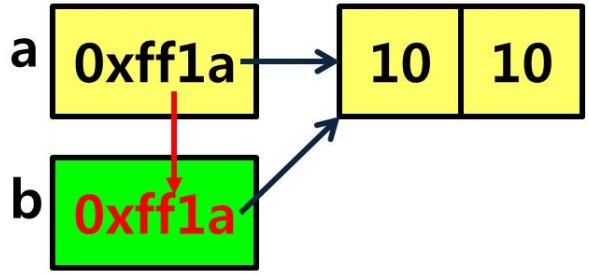
List as input

```
def f(b): # b = a
    b[1] = 0


a = [10,10]
f(a)
print(a)
```



List as input, but ..

```
def f(b): # b = a
    b = [0,0]


a = [10,10]
f(a)
print(a)
```

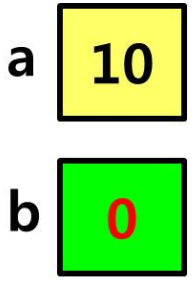


Lists as Function Input (2/2)

- Passing an argument a to function as a parameter b has the effect of b = a (i.e. aliasing)
- For each case, figure out why the output is :

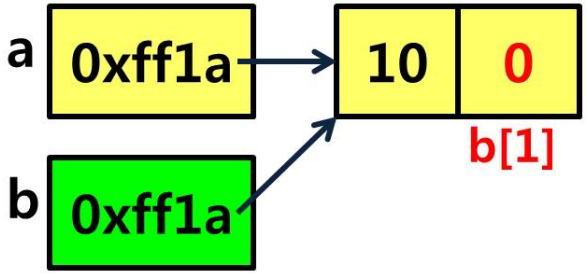
Number as input

```
def f(b):
    b = 0


a = 10
f(a)
print(a)  # 10
# a is not modified
```



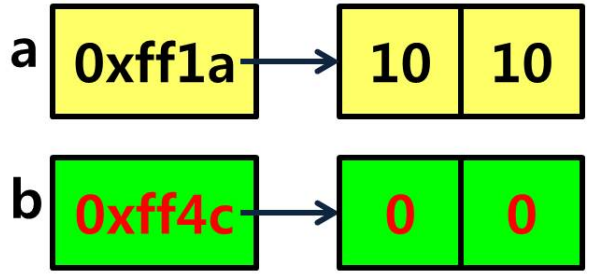
List as input

```
def f(b):
    b[1] = 0


a = [10,10]
f(a)
print(a)  # [10,0]
# a is modified
```



List as input, but ..

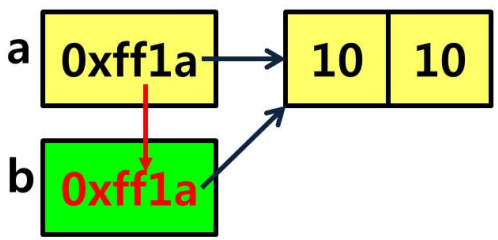
```
def f(b):
    b = [0,0]


a = [10,10]
f(a)
print(a)  # [10,10]
# a is not modified
```



Modifier vs. Pure Function (1/3)

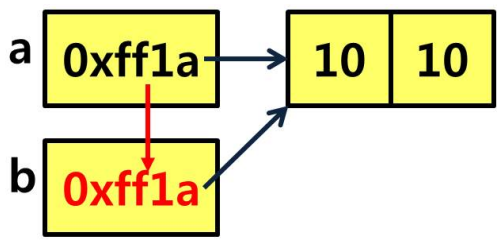
Modifier

```
def f(b): # b = a

    for i in range(len(b)):
        b[i] = b[i] + 1


a = [10,10]
f(a)
```



Pure Function

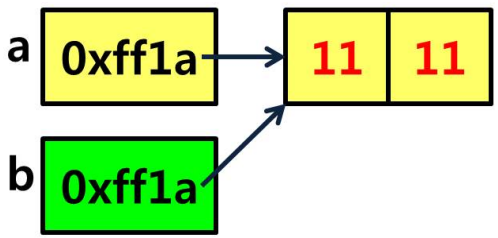
```
def f(b): # b = a
    c = [0]*len(b)
    for i in range(len(b)):
        c[i] = b[i] + 1
    return c

a = [10,10]
d = f(a)
```



Modifier vs. Pure Function (2/3)

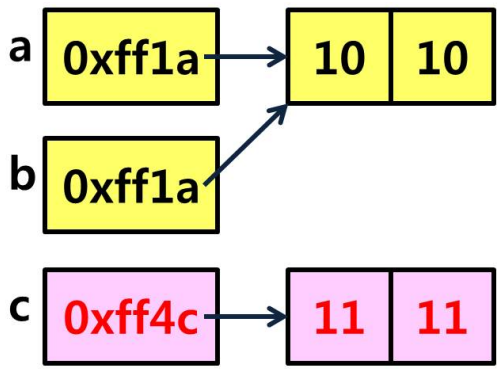
Modifier

```
def f(b):

    for i in range(len(b)):
        b[i] = b[i] + 1


a = [10,10]
f(a)
```



Pure Function

```
def f(b):
    c = [0]*len(b)
    for i in range(len(b)):
        c[i] = b[i] + 1
    return c

a = [10,10]
d = f(a)
```

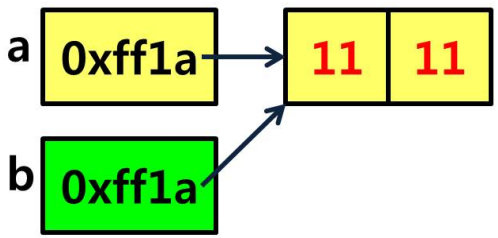


Modifier vs. Pure Function (3/3)

- Receiving return value c to variable d has the effect of d = c

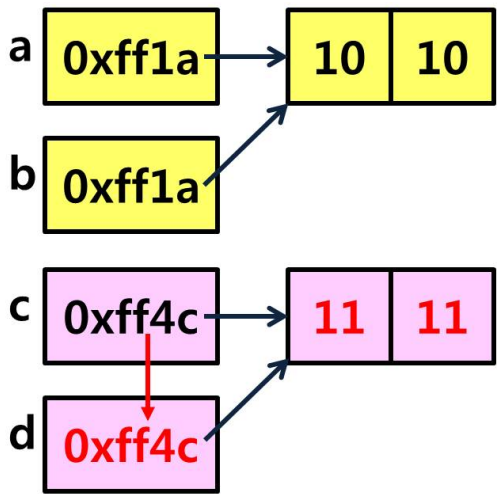
Modifier

```
def f(b):

    for i in range(len(b)):
        b[i] = b[i] + 1


a = [10,10]
f(a)
print(a)  # [11,11]
```



Pure Function

```
def f(b):
    c = [0]*len(b)
    for i in range(len(b)):
        c[i] = b[i] + 1
    return c

a = [10,10]
d = f(a)  # d = c
print(a)  # [10,10]
```



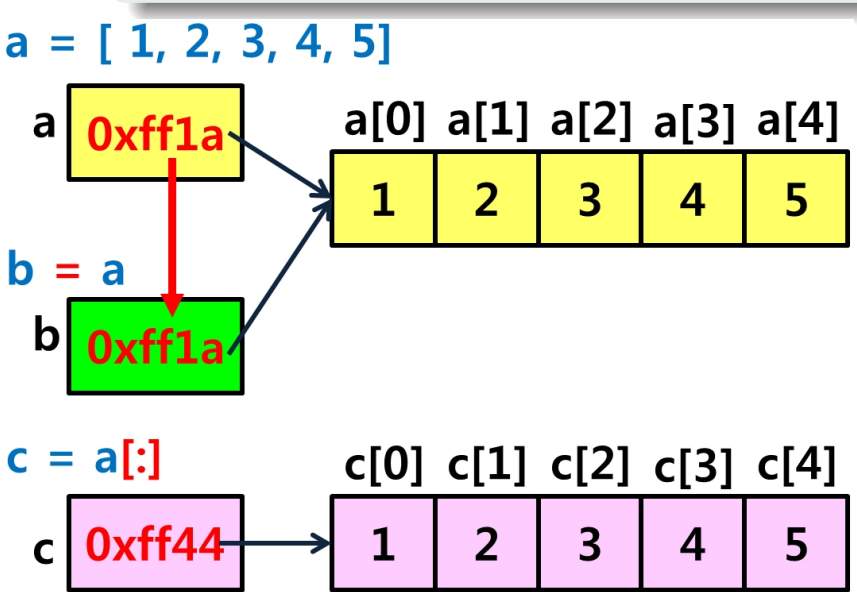
Outline

== vs. is for Lists

- `==` checks if list contents are the same (deep equality)
- `is` checks if aliased (i.e. the same list) (shallow equality)
- `a is b` ⇒ `a == b`, but not vice versa

```
a = [1,2,3,4,5]

b = a
c = a[:]
d = copy.deepcopy(a)
```

```
print(a==b, a==c, a==d)  # True True True
# list contents are the same

print(a is b)  # True   (the same list)
print(a is c)  # False  (different lists)
print(a is d)  # False  (different lists)
```

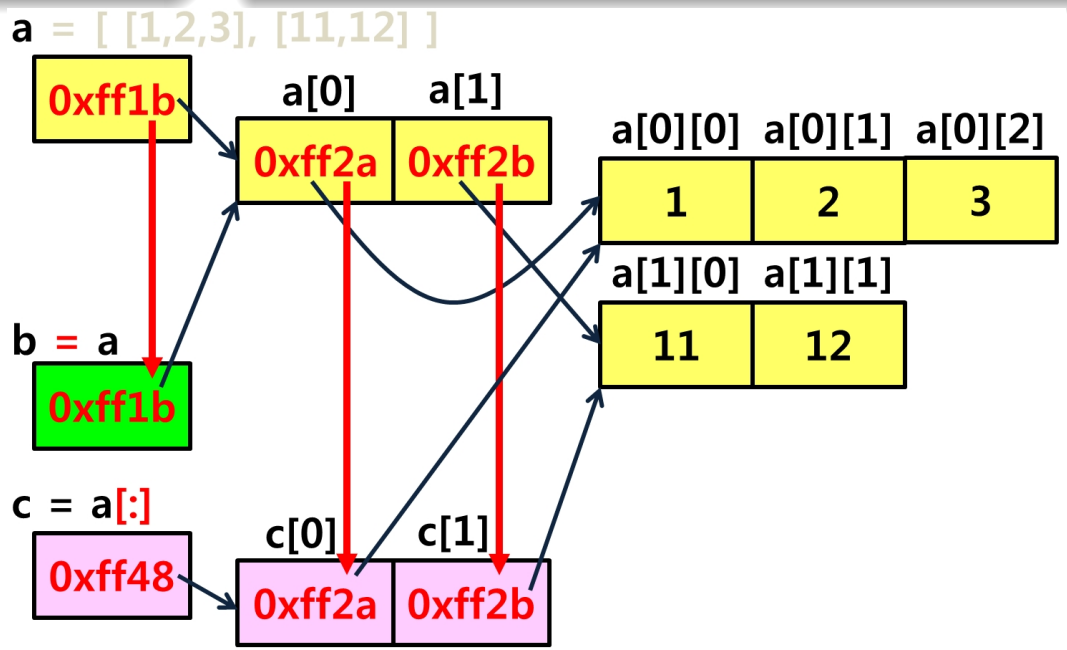


== vs. is for Lists of Lists

- `==` checks if list contents are the same (deep equality)
- `is` checks if aliased (i.e. the same list) (shallow equality)

```
a = [ [1,2,3],
      [4,5] ]

b = a
c = a[:]
d = copy.deepcopy(a)
```

```
print(a==b, a==c, a==d)  # True True True
# list contents are the same

print(a is b)  # True   (the same list)
print(a is c)  # False  (different lists)
print(a is d)  # False  (different lists)
```



Copy/Equality Operations for Lists: Summary

Copy operations for lists

- `=` : alias
  - Beginners mistakenly use alias instead of deep copy
  - Nevertheless, sometimes useful (e.g. practice problems 1-2)
- `[:]` : shallow copy (deep copy of only top-level list)
  - Useful as deep copy for list of numbers (e.g. `[1,2,3]`)
- `copy.deepcopy(·)` : deep copy
  - Useful for multi-dimensional lists (e.g. practice problem 2)

Equality operations for lists

- `is` : check if aliased
  - Seldom used (not used in this course)
- `==` : deep equality
  - Works well with multi-dimensional lists
  - `a is b` $\Rightarrow$ `a == b` , but not vice versa

Outline

Copy/Equality Operations: Lists vs. User-Defined Objects

Recall

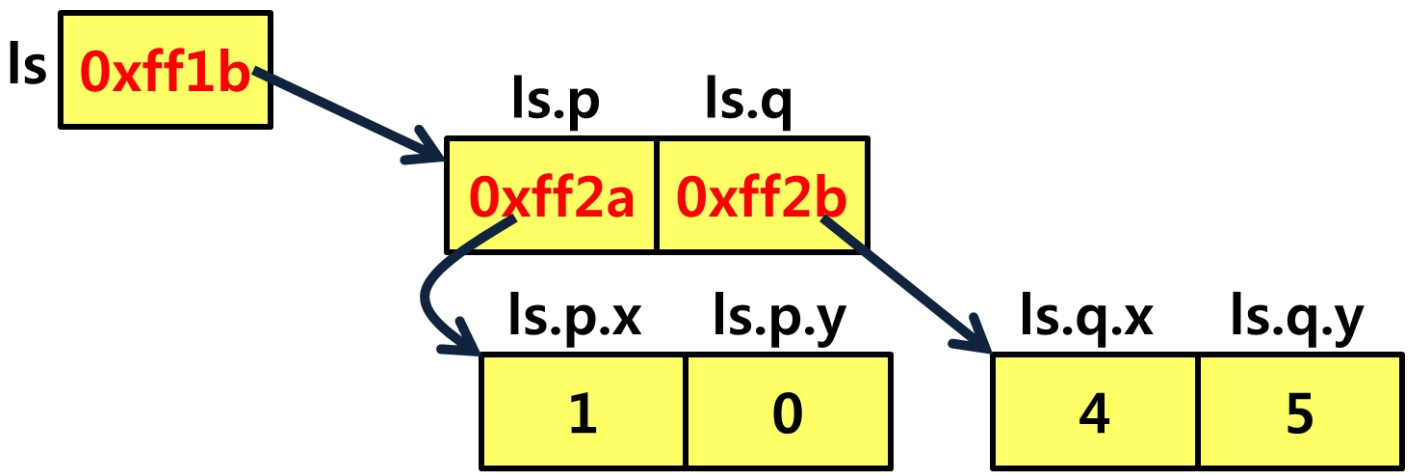
Copy/Equality operations for user-defined objects:

- `=` / `copy.deepcopy(·)` / `is` : the same as for lists
- `==` : $\begin{cases} \text{as you wish (e.g. deep equality)} & \text{if overloaded by } \texttt{\_\_eq\_\_} \\ \texttt{is (check if aliased)} & \text{otherwise} \end{cases}$

- The list is a special case of objects
  - where == is overloaded by __eq__ that recursively checks deep equality
- A user-defined object can be thought of as a list of instance variables
- Thus, copy/equality operations for user-defined objects work almost the same as those for lists

Memory Layout for Objects (simplified for easier understanding)

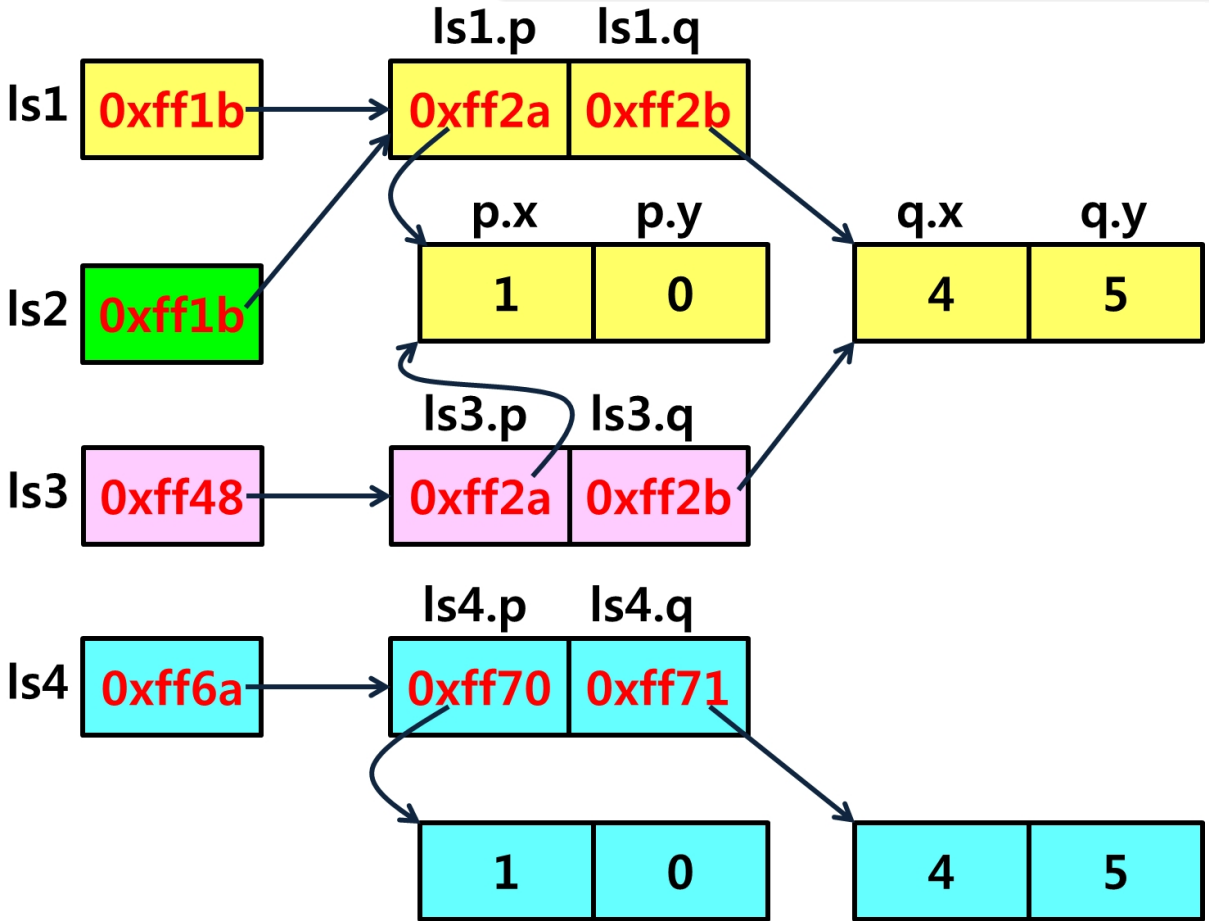
```
class Point:
  def __init__(self,x,y):
    self.x = x
    self.y = y

class LineSegment:
  def __init__(self,point1,point2):
    self.p = point1
    self.q = point2

ls = LineSegment(Point(1,0), Point(4,5))
```



Copy Operations for Objects

```
p = Point(1,0)
q = Point(4,5)
ls1 = LineSegment(p,q)
```

```
ls2 = ls1
ls3 = LineSegment(p,q)
ls4 = copy.deepcopy(ls1)
ls4 = LineSegment(copy.deepcopy(p)
                  ,copy.deepcopy(q))
```

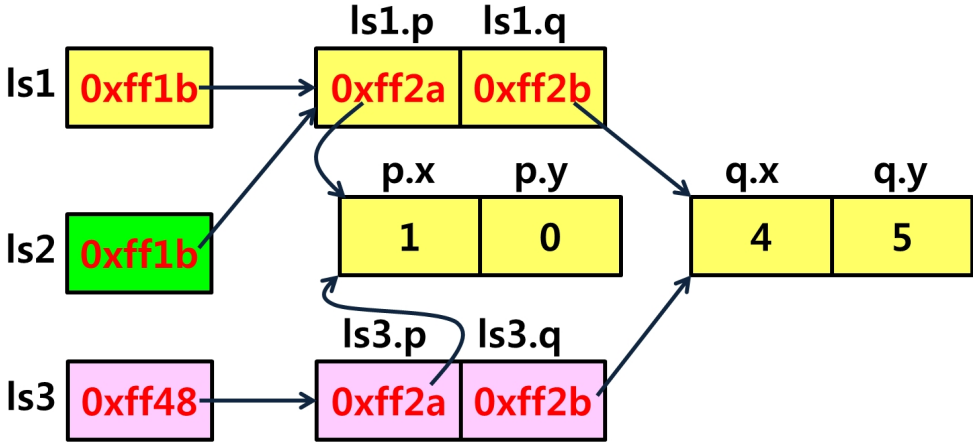


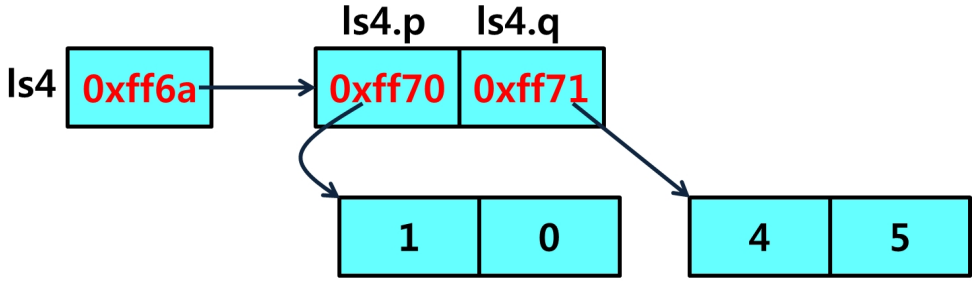
Equality Operations for Objects

- `is` checks if aliased (i.e. the same list) (shallow equality)
- `==` : as you wish (e.g. deep equality) if `__eq__` defined; `is` (check if aliased) otherwise

```
def Point:
  def __eq__(self,p):
    return self.x==p.x and self.y==p.y

def LineSegment:
  def __eq__(self,l):
    return self.p==l.p and self.q==l.q
```

```
print(ls1 == ls2) # True
print(ls1 == ls3) # True
print(ls1 == ls4) # True

print(ls1 is ls2) # True
print(ls1 is ls3) # False
print(ls1 is ls4) # False
```



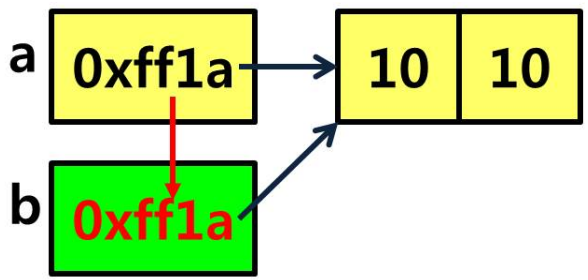


Objects as Function Input (1/2)

- Passing an argument a to function as a parameter b has the effect of b = a (i.e. aliasing)

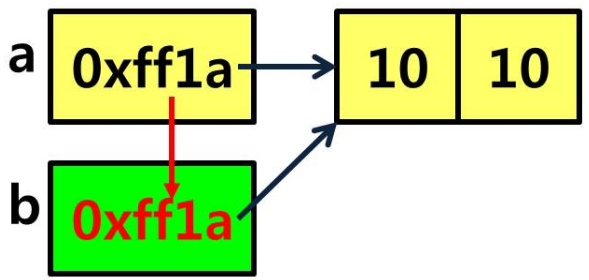
Object as input

```
def f(b): # b = a
    b.y = 0

a = Point(10,10)
f(a)
print(a)
```



Object as input, but ..

```
def f(b): # b = a
    b = Point(0,0)

a = Point(10,10)
f(a)
print(a)
```

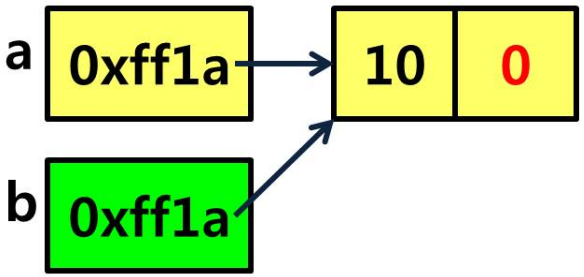


Objects as Function Input (2/2)

- Passing an argument a to function as a parameter b has the effect of b = a (i.e. aliasing)

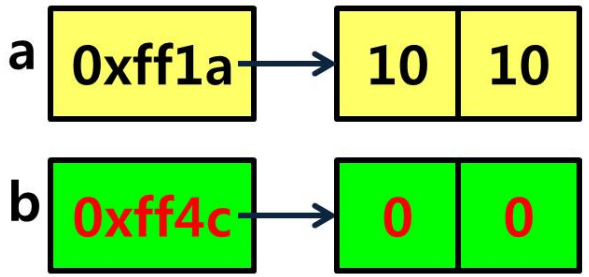
Object as input

```
def f(b):
    b.y = 0


a = Point(10,10)
f(a)
print(a)  # (10,0)
# a is modified
```



Object as input, but ..

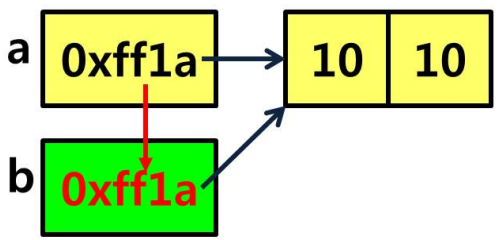
```
def f(b):
    b = Point(0,0)


a = Point(10,10)
f(a)
print(a)  # (10,10)
# a is not modified
```



Modifier vs. Pure Function (1/3)

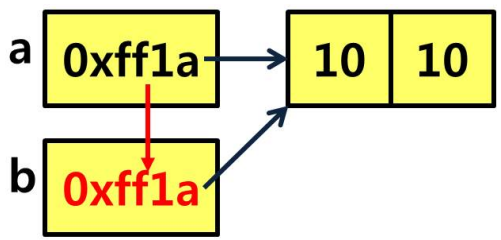
Modifier

```
def f(b): # b = a

    b.x = b.x + 1
    b.y = b.y + 1


a = Point(10,10)
f(a)
```



Pure Function

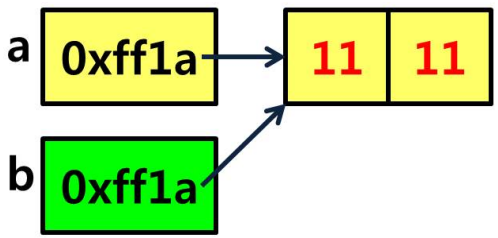
```
def f(b): # b = a
    c = Point(0,0)
    c.x = b.x + 1
    c.y = b.y + 1
    return c

a = Point(10,10)
d = f(a)
```



Modifier vs. Pure Function (2/3)

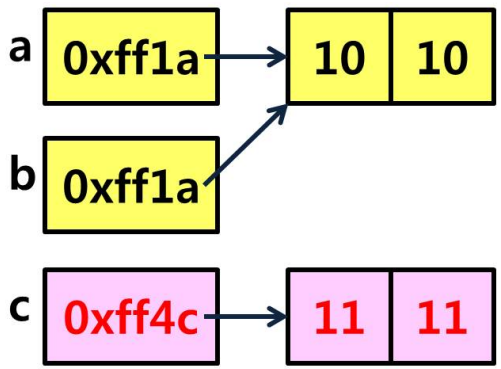
Modifier

```
def f(b):

    b.x = b.x + 1
    b.y = b.y + 1


a = Point(10,10)
f(a)
```



Pure Function

```
def f(b):
    c = Point(0,0)
    c.x = b.x + 1
    c.y = b.y + 1
    return c

a = Point(10,10)
d = f(a)
```

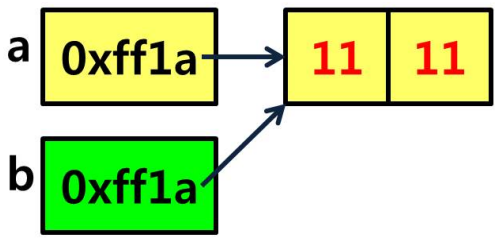


Modifier vs. Pure Function (3/3)

- Receiving return value c to variable d has the effect of d = c

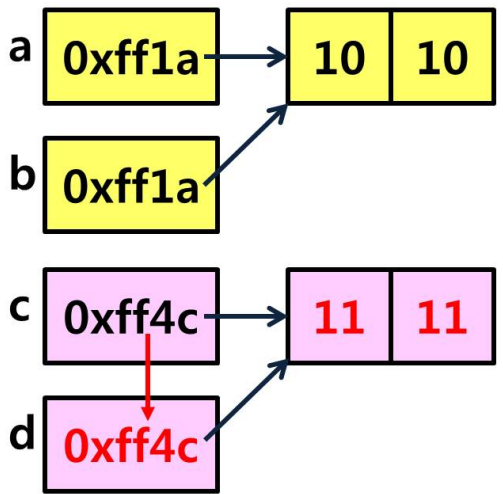
Modifier

```
def f(b):

    b.x = b.x + 1
    b.y = b.y + 1


a = Point(10,10)
f(a)
print(a)   # (11,11)
```



Pure Function

```
def f(b):
    c = Point(0,0)
    c.x = b.x + 1
    c.y = b.y + 1
    return c

a = Point(10,10)
d = f(a)  # d = c
print(a,d)   # (10,10) (11,11)
```



Modifiers for Lists?

`L.sort()`/`L.append()`/.. are really modifiers for lists?

- Think of lists as user-defined objects of the class `List`
- Then, `L.sort()` can be thought of `List.sort(L)` where `L` is pass as an argument to modifier `List.sort`, and sorting performs directly on `L`
- On the contrary, list concatenation operation `+` can be understood as a pure function

```
def f(L):
    L.append(40)

L = [10,20,30]
f(L)
print(L)
```

Figure out why the output is:

```
[10, 20, 30, 40]
```