

16. More On Lists

Topics:

References

Alias

More on Slicing

Merging Sorted Lists

Comparing Lists

You can use `==` to compare two lists

```
>>> x = [10,20,30,40]
>>> y = [10,20,30,40]
>>> x==y
True
```

Comparing Lists

You can use `==` to compare two lists

x -->

0	---->	10
1	---->	20
2	---->	30
3	---->	40

y -->

0	---->	10
1	---->	20
2	---->	30
3	---->	40

The Boolean expression `x==y` is True because x and y have the same length and identical values in each element

Comparing Lists

You can use `==` to compare two lists

```
>>> x = [1,2,3]
>>> y = [1.0,2.0,3.0]
>>> x==y
True
```

If there are ints and floats, convert everything to float then compare

Comparing Lists

Do not use `<`, `<=`, `>`, `>=` to compare two lists

```
>>> x = [10, 20, 30, 40]
>>> y = [11, 21, 31, 41]
>>> x < y
True
>>> y < x
True
```

Unpredictable

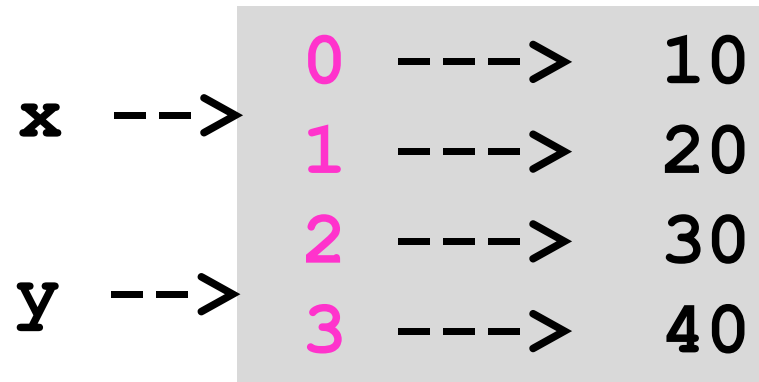
Aliasing

This:

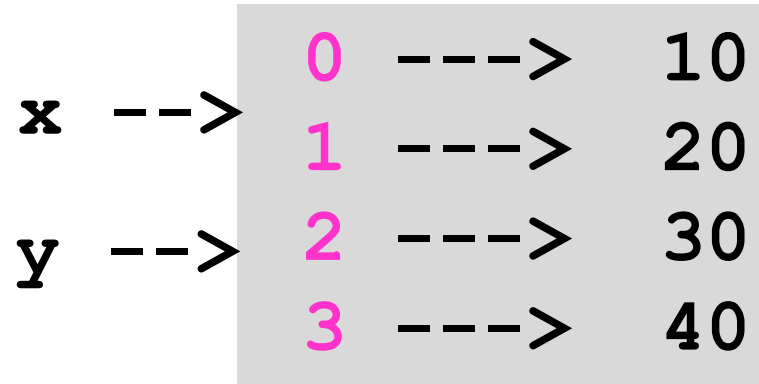
```
x = [10, 20, 30, 40]
```

```
y = x
```

Results in this:



Aliasing



Things to say:

x and y are variables that refer to the same list object.

The object is aliased because it has more than one name.

Tracking Changes

● $\mathbf{x} = [10, 20, 30, 40]$

$\mathbf{y} = \mathbf{x}$

$\mathbf{y} = [1, 2, 3]$

$\mathbf{x} \dashrightarrow$

0	\dashrightarrow	10
1	\dashrightarrow	20
2	\dashrightarrow	30
3	\dashrightarrow	40

Tracking Changes

x = [10, 20, 30, 40]

● **y** = **x**

y = [1, 2, 3]

x -->

y -->

0	---	>	10
1	---	>	20
2	---	>	30
3	---	>	40

Tracking Changes

$\mathbf{x} = [10, 20, 30, 40]$

$\mathbf{y} = \mathbf{x}$

● $\mathbf{y} = [1, 2, 3]$

$\mathbf{x} \dashrightarrow$

0	---	>	10
1	---	>	20
2	---	>	30
3	---	>	40

$\mathbf{y} \dashrightarrow$

0	---	>	1
1	---	>	2
2	---	>	3

The is Operator

```
>>> x = [10, 20, 30, 40]
>>> y = [10, 20, 30, 40]
>>> x is y
False
```

x --->

0	---	>	10
1	---	>	20
2	---	>	30
3	---	>	40

y --->

0	---	>	10
1	---	>	20
2	---	>	30
3	---	>	40

Even though the two lists have the same component values. *x* and *y* do not refer to the same object.

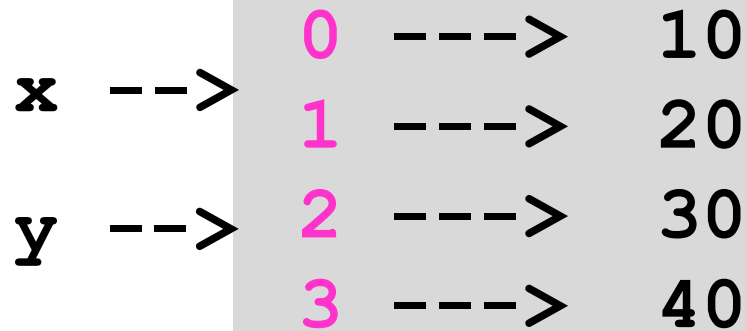
The is Operator

```
>>> x = [10, 20, 30, 40]
```

```
>>> y = x
```

```
>>> x is y
```

```
True
```



x and y refer to the same object

Making a Copy of a List

● `x = [10, 20, 30, 40]`
`y = list(x)`

`x` --->

0	---	>	10
1	---	>	20
2	---	>	30
3	---	>	40

Making a Copy of a List

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

●

```
y = list(x)
```

```
x --->
```

0	---	>	10
1	---	>	20
2	---	>	30
3	---	>	40

```
y --->
```

0	---	>	10
1	---	>	20
2	---	>	30
3	---	>	40

Slices Create new Objects

● `x = [10, 20, 30, 40]`
`y = x[1:]`

`x` --->

0	---	>	10
1	---	>	20
2	---	>	30
3	---	>	40

Slices Create New Objects

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

●

```
y = x[1:]
```

x --->

0	---	>	10
1	---	>	20
2	---	>	30
3	---	>	40

y --->

0	---	>	20
1	---	>	30
2	---	>	40

Careful!

● `x = [10, 20, 30, 40]`

`y = x`

`y = x[1:]`

`x` --->

0	---	>	10
1	---	>	20
2	---	>	30
3	---	>	40

Careful!

x = [10, 20, 30, 40]

y = **x**

y = **x**[1:]

x -->

y -->

0	---	>	10
1	---	>	20
2	---	>	30
3	---	>	40

Careful!

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

```
y = x
```

```
● y = x[1:]
```

```
x --->
```

0	---	>	10
1	---	>	20
2	---	>	30
3	---	>	40

```
y --->
```

0	---	>	20
1	---	>	30
2	---	>	40

Void Functions

● `x = [40, 20, 10, 30]`
`y = x.sort()`

`x` -->

0	---	>	40
1	---	>	20
2	---	>	10
3	---	>	30

`y` -->

Void Functions

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

●

```
y = x.sort()
```

```
x -->
```

0	---	>	10
1	---	>	20
2	---	>	30
3	---	>	40

```
y --> None
```

Void Functions return None, a special type

Void Functions

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

●

```
y = list(x)
```

```
y.sort()
```

```
x -->
```

0	---	>	40
1	---	>	20
2	---	>	10
3	---	>	30

```
y -->
```

0	---	>	40
1	---	>	20
2	---	>	10
3	---	>	30

Void Functions return None, a special type

Void Functions

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

```
y = list(x)
```

●

```
y.sort()
```

x --->

0	---	>	40
1	---	>	20
2	---	>	10
3	---	>	30

y --->

0	---	>	10
1	---	>	20
2	---	>	30
3	---	>	40

Void Functions return None, a special type

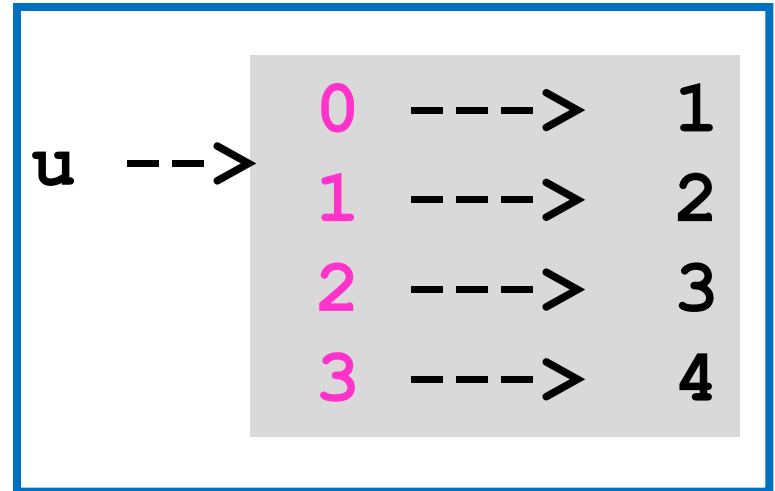
Understanding Function Calls

```
def f(x):  
    x = x[1:]  
    print x  
  
if __name__ == '__main__':  
    u = [1,2,3,4]  
    f(u)  
    print u
```

Looks like f deletes the 0-th character in x

Understanding Function Calls

```
def f(x):  
    x = x[1:]  
    print x  
  
if __name__ == '__blabla'  
    ● u = [1, 2, 3, 4]  
    f(u)  
    print u
```

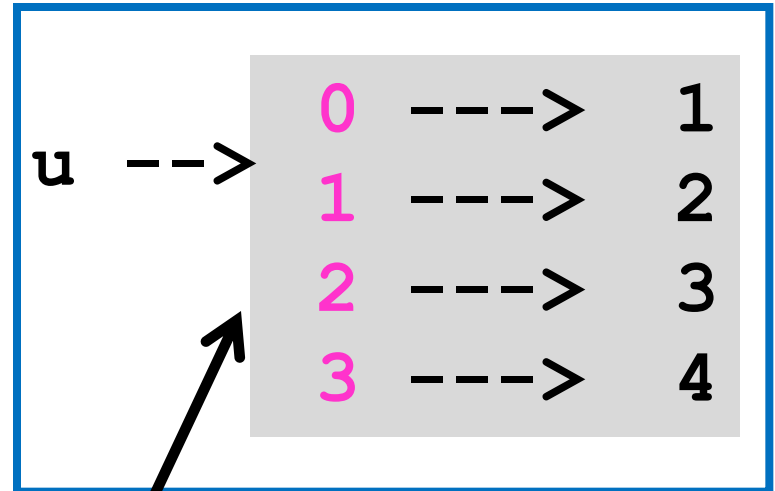


Follow the red dot and watch for impact...

Understanding Function Calls

```
def f(x):  
    x = x[1:]  
    print x  
  
if __name__ == '__blabla':  
    u = [1, 2, 3, 4]  
    f(u)  
    print u
```

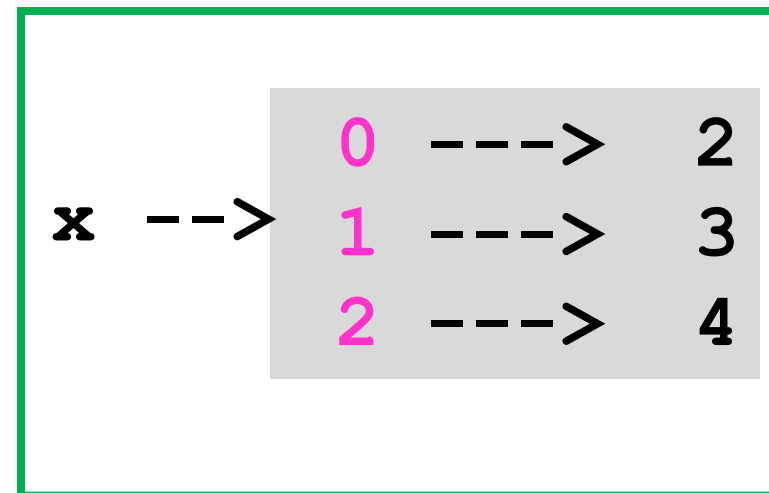
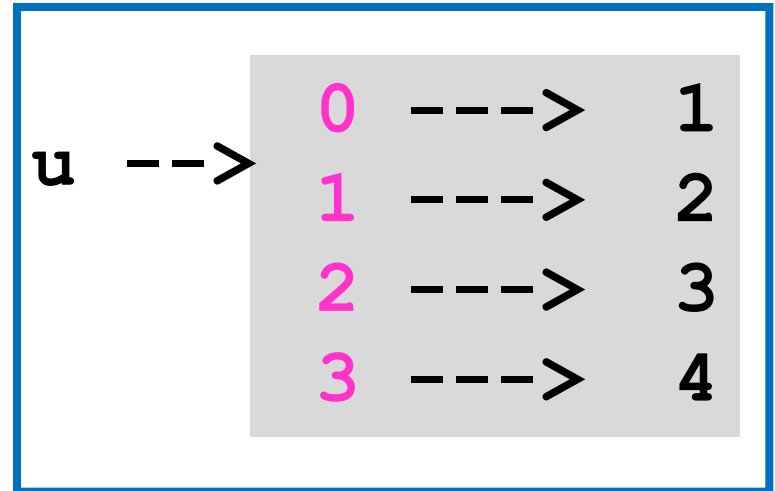
Parameter x initially refers to the same object as u



Understanding Function Calls

```
def f(x):  
    ● x = x[1:]  
    print x  
  
if __name__ == '__blabla'  
    u = [1,2,3,4]  
    f(u)  
    print u
```

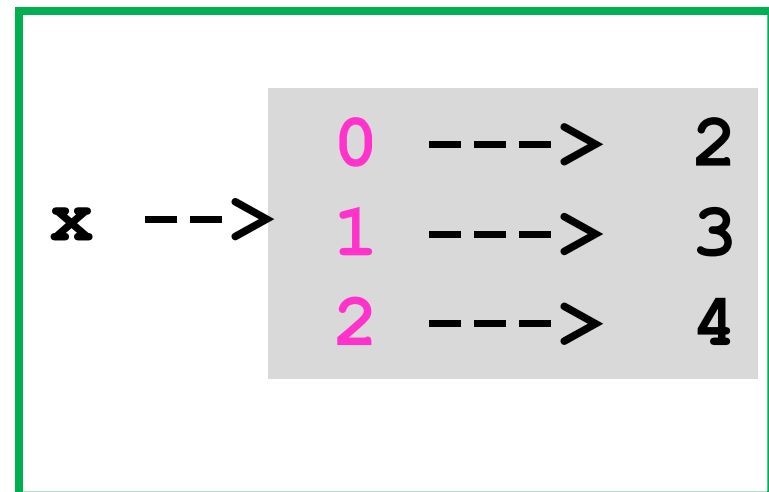
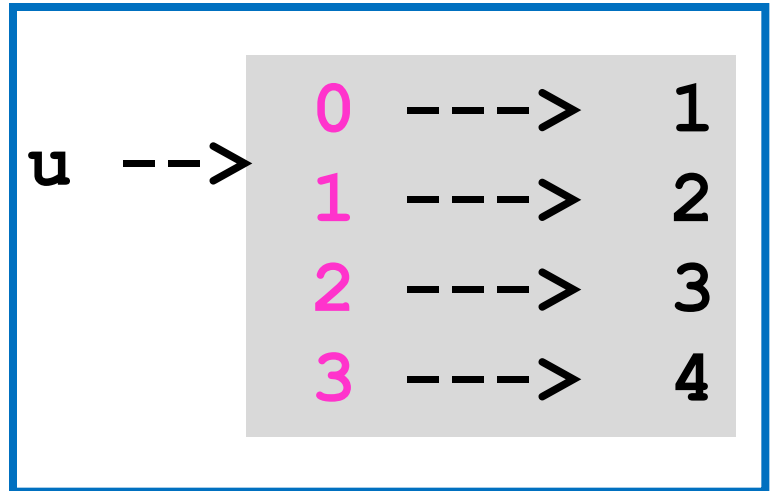
x[1:] creates a new object
and x will refer to it



Understanding Function Calls

```
def f(x):  
    x = x[1:]  
    ● print x  
  
if __name__ == '__blabla':  
    u = [1, 2, 3, 4]  
    f(u)  
    print u
```

2 3 4 is printed



Understanding Function Calls

```
def f(x):  
    x = x[1:]  
    print x  
  
if __name__ == '__blabla':  
    u = [1,2,3,4]  
    f(u)  
    ● print u
```

1 2 3 4 is printed

u --->

0	---	>	1
1	---	>	2
2	---	>	3
3	---	>	4

x --->

0	---	>	2
1	---	>	3
2	---	>	4

Some Inadvertent Errors

```
>>> x = [10,20,30]
>>> y = [11,21,31]
>>> z = x+y
>>> print z
[10,20,30,11,21,31]
```

Some Inadvertent Errors

```
>>> x = [10,20,30]
>>> y = 3*x
>>>Print y
[10,20,30,10,20,30,10,20,3]
```

List Comprehensions

A short cut for setting up "simple" lists

```
>>> x = [i for i in range(5)]  
>>> print x  
[0, 1, 2, 3, 4]
```


List Comprehensions

A short cut for setting up "simple" lists

```
>>> x = [1 for i in range(5)]  
>>> print x  
[1,1,1,1,1]
```

List Comprehensions

A short cut for setting up "simple" lists

```
>>> x = [math.sqrt(i) for i in range(5)]  
>>> print x  
[0, 1, 1.414, 1.732, 2.0]
```

Quickly: Lists of Strings

```
>>> x = ['Maine', 'Vermont', 'New York']
>>> a = x[1]
>>> print a
'Vermont'
>>> c = a[2]
>>> print c
'r'
>>> x[1][2]
'r'
```

Next Problem

Merging Two Sorted Arrays
Into a
Single Sorted Array

Example

x ->

12	33	35	45
----	----	----	----

y ->

15	42	55	65	75
----	----	----	----	----

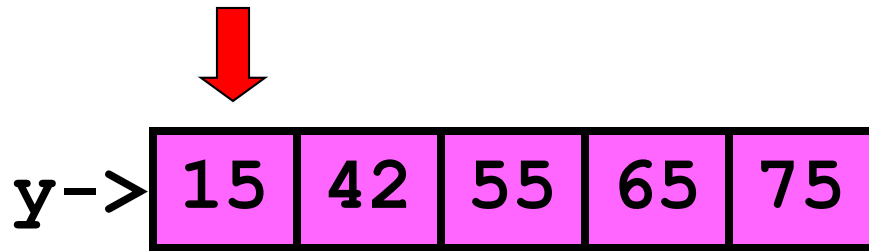
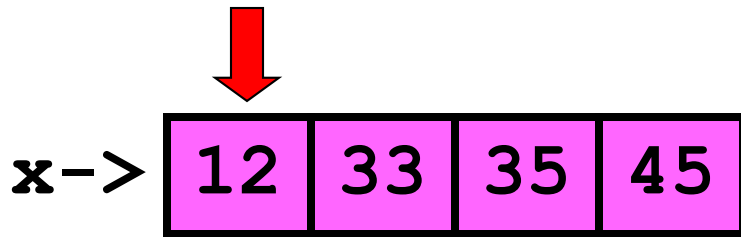
x and y are input
They are sorted

z is the output

z ->

12	15	33	35	42	45	55	65	75
----	----	----	----	----	----	----	----	----

Merging Two Sorted Lists



z → []

ix and iy
keep track
of where
we are in x
and y

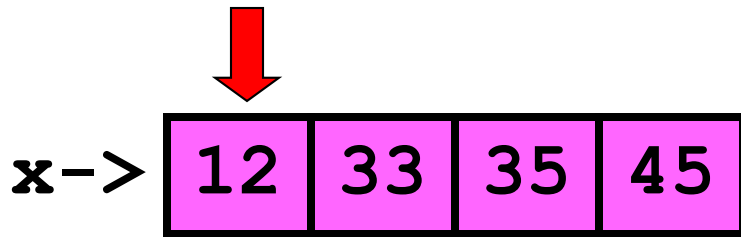
ix:

0

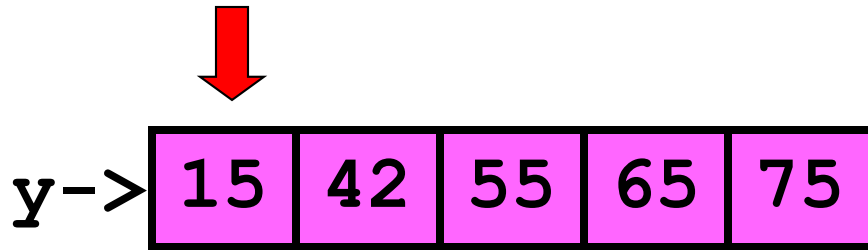
iy:

0

Merging Two Sorted Lists



ix: [0]



iy: [0]

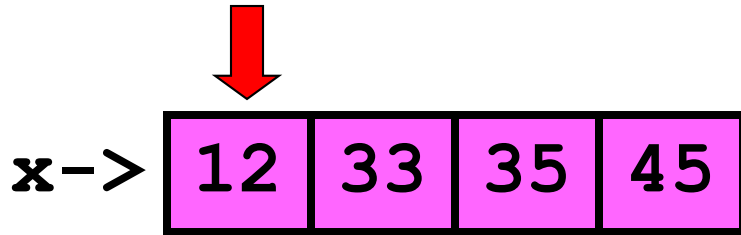
z -> []

Do we pick from **x** ?

$x[ix] \leq y[iy]$

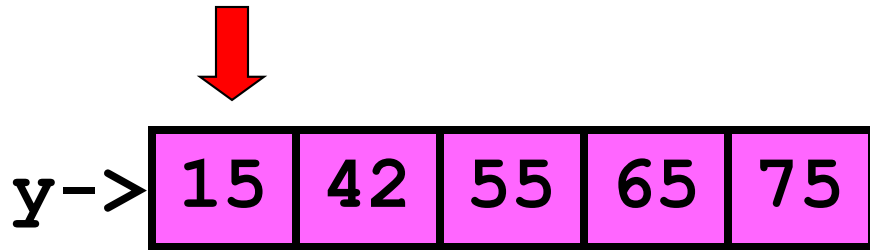
???

Merge



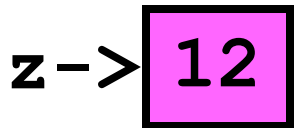
ix:

0



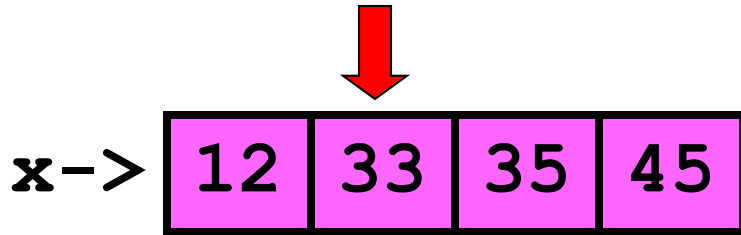
iy:

0



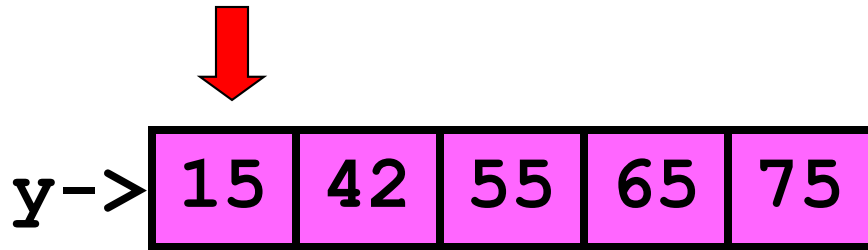
Yes. So update ix

Merge



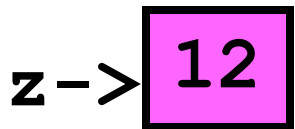
ix:

1



iy:

0

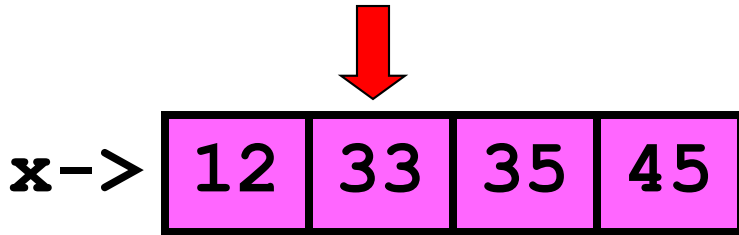


Do we pick from x ?

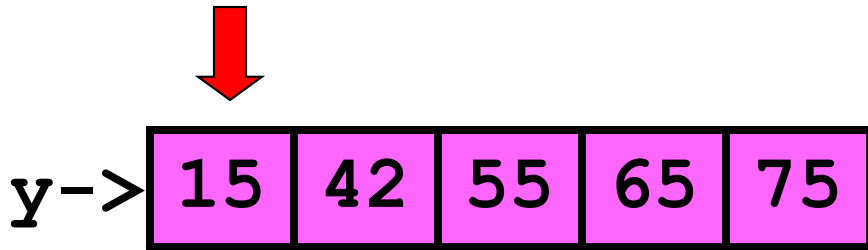
$x[ix] \leq y[iy]$

???

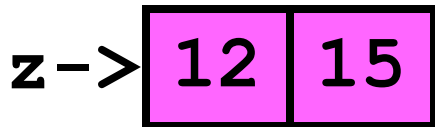
Merge



ix: [1]



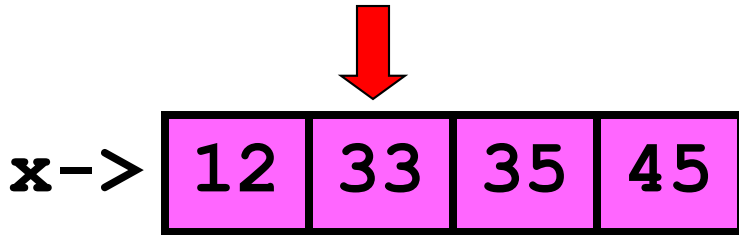
iy: [0]



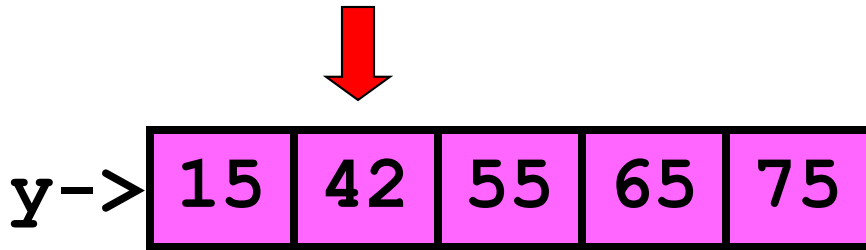
iz:

No. So update iy

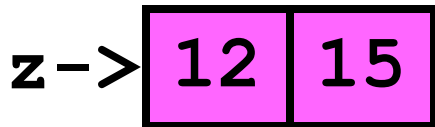
Merge



ix: [1]



iy: [1]

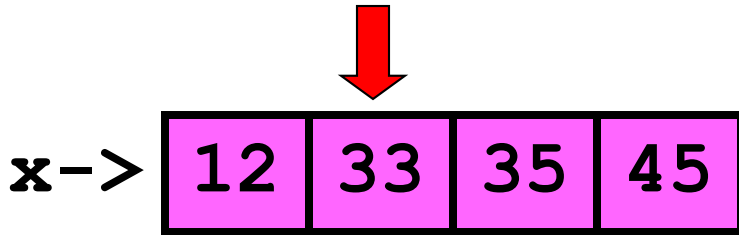


Do we pick from x ?

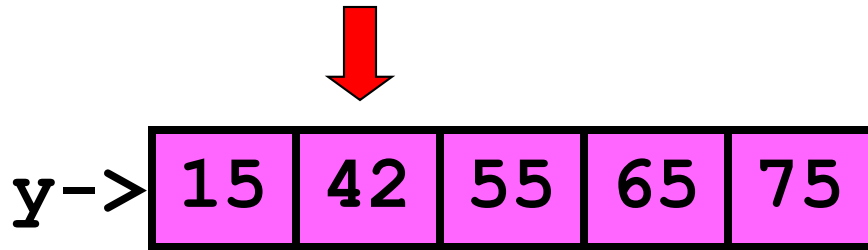
$x[ix] \leq y[iy]$

???

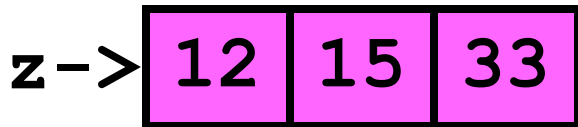
Merge



ix: [1]

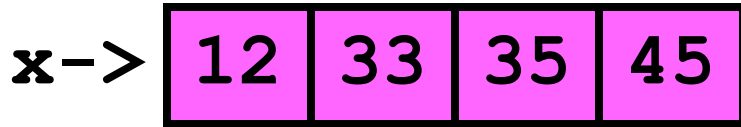


iy: [1]



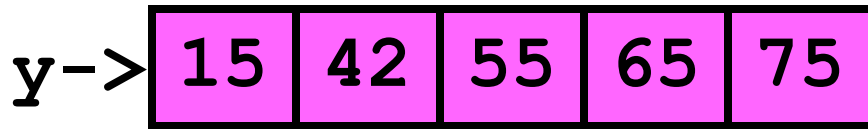
Yes. So update ix

Merge



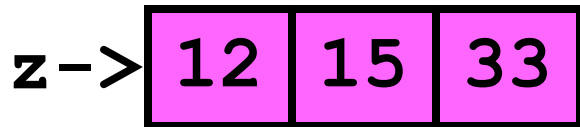
ix:

2



iy:

1

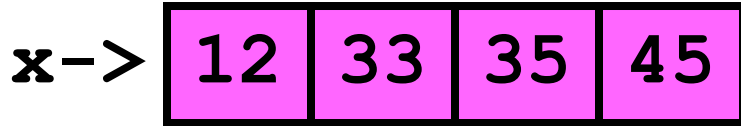


Do we pick from x ?

$x[ix] \leq y[iy]$

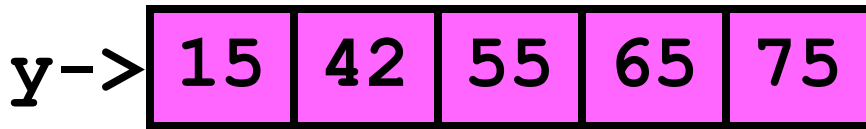
???

Merge



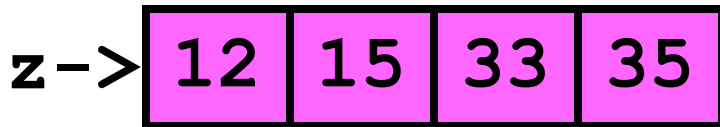
ix:

2



iy:

1



Yes. So update ix

Merge



x ->

12	33	35	45
----	----	----	----

ix:

3



y ->

15	42	55	65	75
----	----	----	----	----

iy:

1

z ->

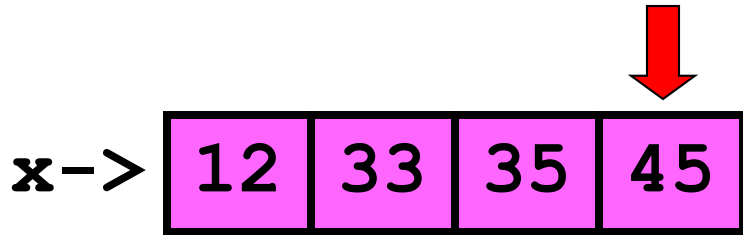
12	15	33	35
----	----	----	----

Do we pick from x ?

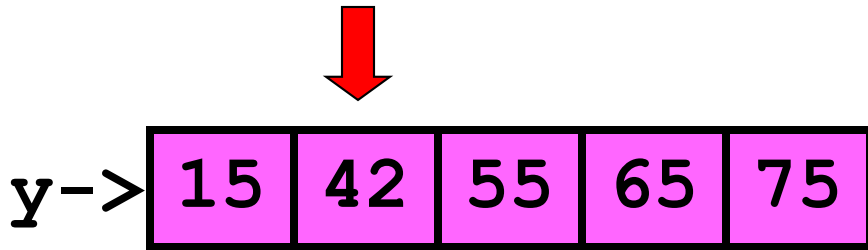
$x[ix] \leq y[iy]$

???

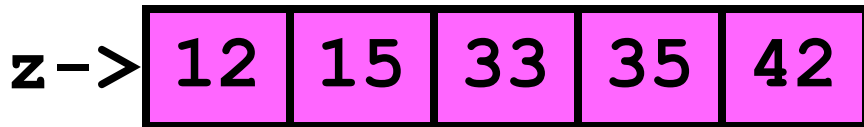
Merge



ix: [3]

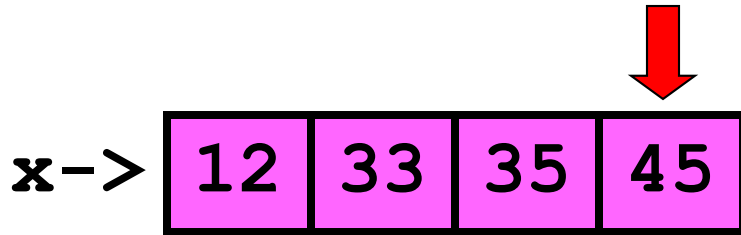


iy: [1]



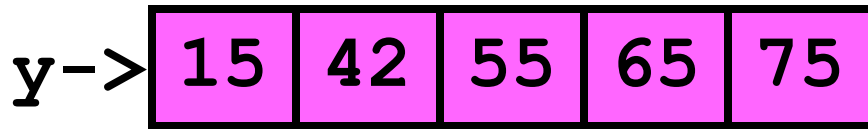
No. So update iy...

Merge



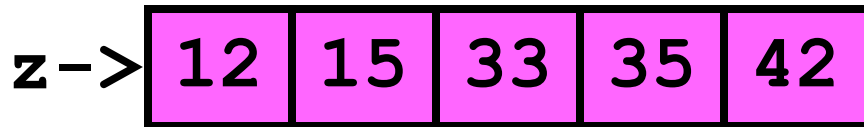
ix:

3



iy:

2

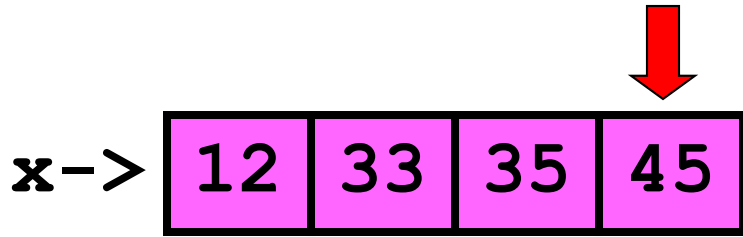


Do we pick from x ?

$x[ix] \leq y[iy]$

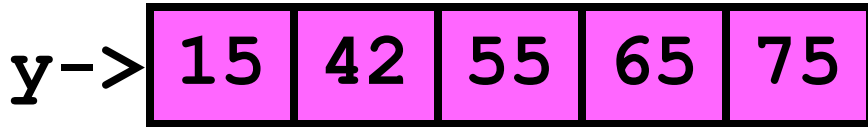
???

Merge



ix:

3



iy:

2



Yes. So update ix.

Merge



x ->

12	33	35	45
----	----	----	----

ix:

4



y ->

15	42	55	65	75
----	----	----	----	----

iy:

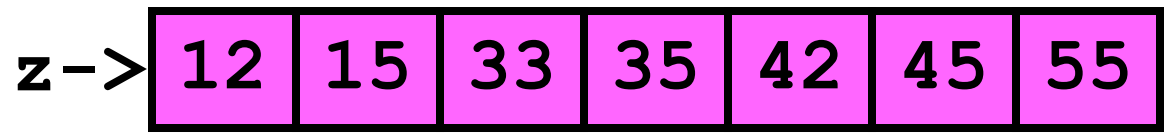
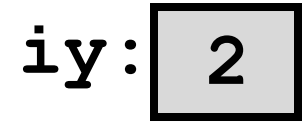
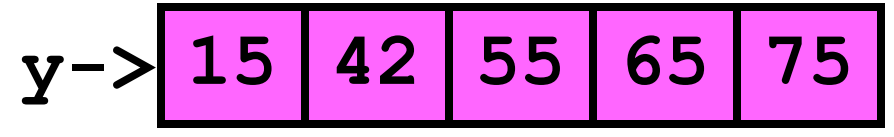
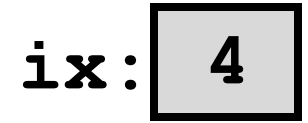
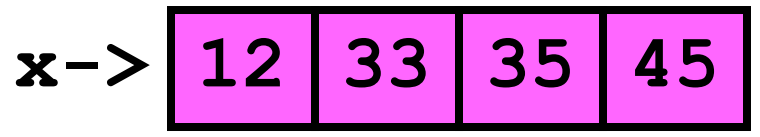
2

z ->

12	15	33	35	42	45
----	----	----	----	----	----

Done with x. Pick from y

Merge



So update iy

Merge



x ->

12	33	35	45
----	----	----	----

ix:

4



y ->

15	42	55	65	75
----	----	----	----	----

iy:

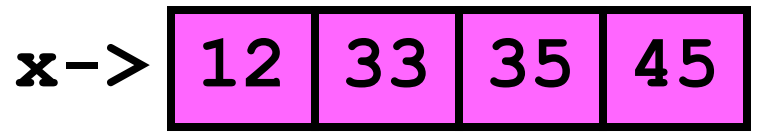
3

z ->

12	15	33	35	42	45	55
----	----	----	----	----	----	----

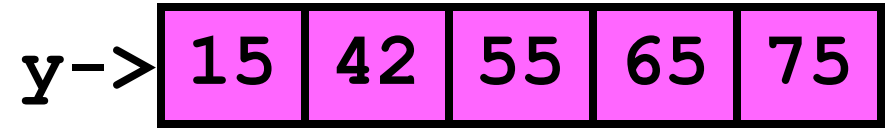
Done with x. Pick from y

Merge



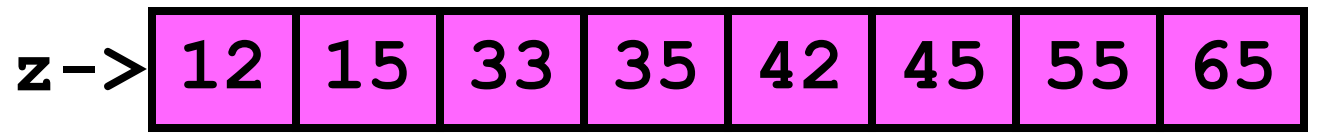
ix:

4



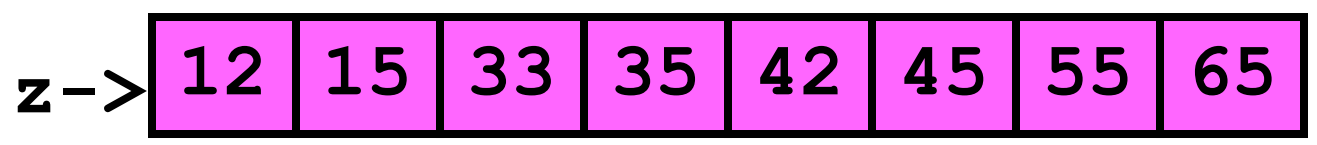
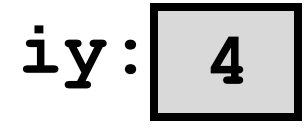
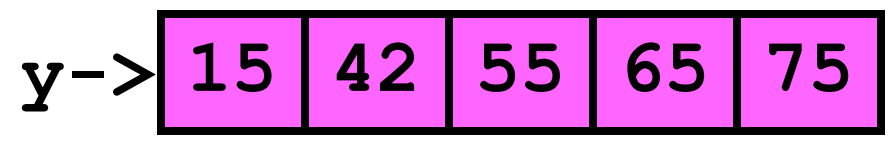
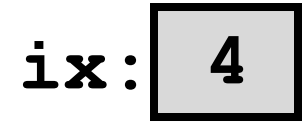
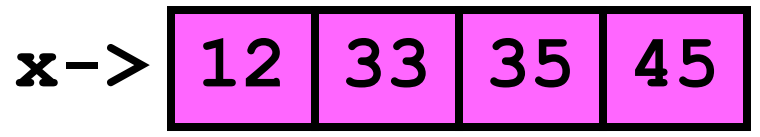
iy:

3



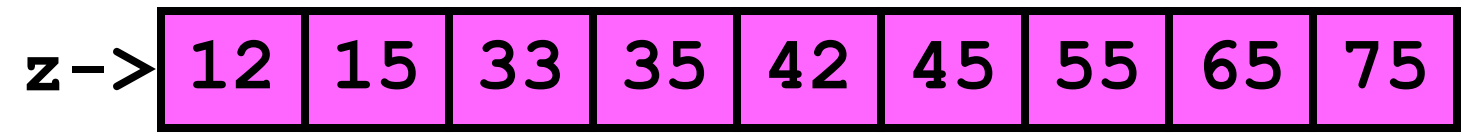
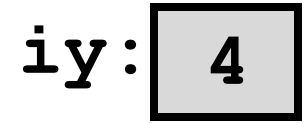
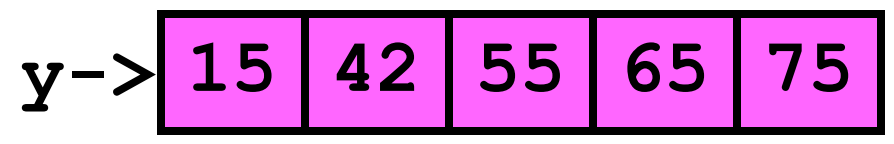
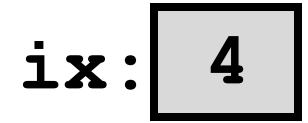
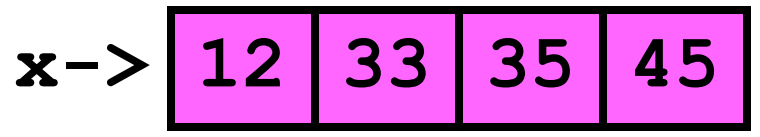
So update iy.

Merge



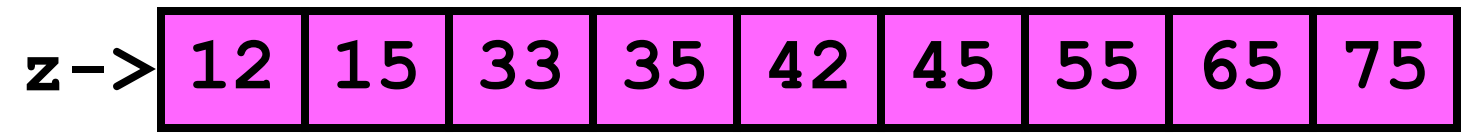
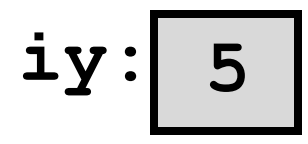
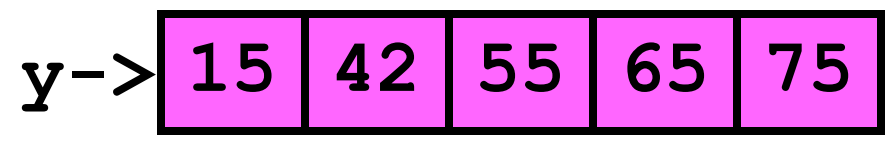
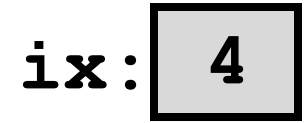
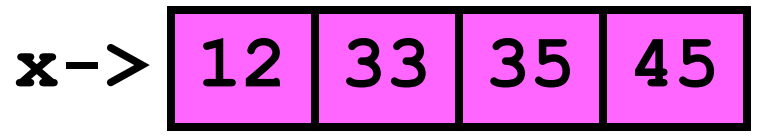
Done with x. Pick from y

Merge



Update iy

Merge



All Done

The Python Implementation...

```

def Merge(x, y):
    n = len(x); m = len(y);
    ix = 0; iy = 0; z = []
    for iz in range(n+m):
        if ix >= n:
            z.append(y[iy]); iy += 1
        elif iy >= m:
            z.append(x[ix]); ix += 1
        elif x[ix] <= y[iy]:
            z.append(x[ix]); ix += 1
        elif x[ix] > y[iy]:
            z.append(y[iy]); iy += 1
    return z

```

Build z up
via repeated
appending

x-list exhausted y-list exhausted x-value smaller y-value smaller

```

def Merge(x, y):
    n = len(x); m = len(y);
    ix = 0; iy = 0; z = []
    for iz in range(n+m):
        if ix >= n:
            z.append(y[iy]); iy += 1
        elif iy >= m:
            z.append(x[ix]); ix += 1
        elif x[ix] <= y[iy]:
            z.append(x[ix]); ix += 1
        elif x[ix] > y[iy]:
            z.append(y[iy]); iy += 1
    return z

```

len(x)+len(y)
is the total length
of the merged list

x-list exhausted y-list exhausted x-value smaller y-value smaller

Implementation Using Pop

```
def Merge(x,y):  
    u = list(x)           Make copies of the  
    v = list(y)           Incoming lists  
    z = []  
    while len(u)>0 and len(v)>0 :  
        if u[0]<= v[0]:  
            g = u.pop(0)  
        else:  
            g = v.pop(0)  
        z.append(g)  
    z.extend(u)  
    z.extend(v)  
    return z
```

Implementation Using Pop

```
def Merge(x,y):  
    u = list(x)  
    v = list(y)  
    z = []  
    while len(u)>0 and len(v)>0 :  
        if u[0]<= v[0]:  
            g = u.pop(0)  
        else:  
            g = v.pop(0)  
        z.append(g)  
    z.extend(u)  
    z.extend(v)  
    return z
```

Build z up via repeated appending

Implementation Using Pop

```
def Merge(x,y):  
    u = list(x)  
    v = list(y)  
    z = []  
    while len(u)>0 and len(v)>0 :  
        if u[0]<= v[0]:  
            g = u.pop(0)  
        else:  
            g = v.pop(0)  
        z.append(g)  
    z.extend(u)  
    z.extend(v)  
    return z
```

Every "pop" reduces the length by 1. The loop shuts down when one of u or v is exhausted

Implementation Using Pop

```
def Merge(x,y):  
    u = list(x)  
    v = list(y)  
    z = []  
    while len(u)>0 and len(v)>0 :  
        if u[0]<= v[0]:  
            g = u.pop(0)  
        else:  
            g = v.pop(0)  
        z.append(g)  
    z.extend(u)  
    z.extend(v)  
    return z
```

g gets the popped value
and it is appended to z

Implementation Using Pop

```
def Merge(x,y):
    u = list(x)
    v = list(y)
    z = []
    while len(u)>0 and len(v)>0 :
        if u[0]<= v[0]:
            g = u.pop(0)
        else:
            g = v.pop(0)
        z.append(g)
    z.extend(u)
    z.extend(v)
    return z
```

Add what is left in u.
OK if u is the empty list

Implementation Using Pop

```
def Merge(x,y):  
    u = list(x)  
    v = list(y)  
    z = []  
    while len(u)>0 and len(v)>0 :  
        if u[0]<= v[0]:  
            g = u.pop(0)  
        else:  
            g = v.pop(0)  
        z.append(g)  
    z.extend(u)  
    z.extend(v)  
    return z
```

Add what is left in v.
OK if v is the empty list