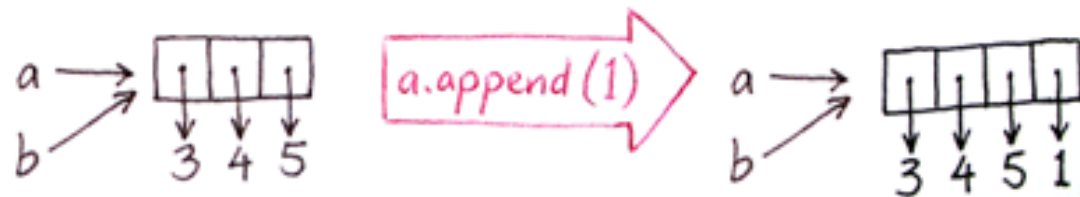
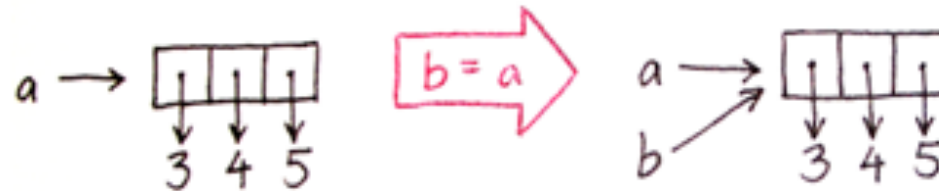


15-112

Fundamentals of Programming

Week 4 - Lecture 1: Lists



Builtin Data Types

Python name

Description

Values

NoneType	absence of value	None
bool (boolean)	Boolean values	True, False
int (integer)	integer values	-2^{63} to $2^{63} - 1$
long	large integer values	all integers
float	fractional values	e.g. 3.14
complex	complex values	e.g. 1+5j
str (string)	text	e.g. "Hello World!"
list	a list of values	e.g. [2, "hi", 3.14]

...

String vs List

string

```
s = "hw3 was easy"
```

A sequence (string)
of characters.

immutable

~~s[0] = "H"~~

list

```
a = [1, 3.14, "hi", True]
```

A sequence of
arbitrary objects.

mutable

```
a[0] = 100
```

String vs List

string

```
s = "hw3 was easy"
```

A sequence (string)
of characters.

immutable

```
s[0] = "H"
```

list

```
a = [1, 3.14, "hi", True]
```

A sequence of
arbitrary objects.

mutable

```
a[0] = 100
```

Lists: basic usage

`a = []` **# creates an empty list**

`b = list()` **# also creates an empty list**

`c = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

`d = list(range(1, 11))` `d = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

`e = [1, 3.14, None, True, "Hi", [1, 2, 3]]`

for `i` **in** `range(len(c))`:
 `print(c[i])`

for `item` **in** `e`:
 `print(item)`

`print(e[1:4])`

`e[2] = 0`

`print(e[::-2])`

Lists: basic usage

```
print([1, 2, 3] + [4, 5, 6])
```

```
a = [0] * 5
```

```
print(a)
```

```
if (1 in a):
```

```
    print("1 is in the list a.")
```

```
if (1 not in a):
```

```
    print("1 is not in the list a.")
```

```
b = [0, 0, 0, 0, 0]
```

```
if (a == b):
```

```
    print("a and b contain the same elements.")
```

Lists: built-in functions

```
a = list(range(1, 11))
```

```
print(len(a))
```

```
print(min(a))
```

```
print(max(a))
```

```
print(sum(a))
```

```
a = [4, 5, 1, 3, 2, 8, 7, 6, 9, 10]
```

```
a = sorted(a)
```

```
print(a)
```

Lists: interesting example

```
x = 1
```

```
y = x
```

```
x = 2
```

```
print(x, y)      2 1
```

```
x = [1, 2, 3]
```

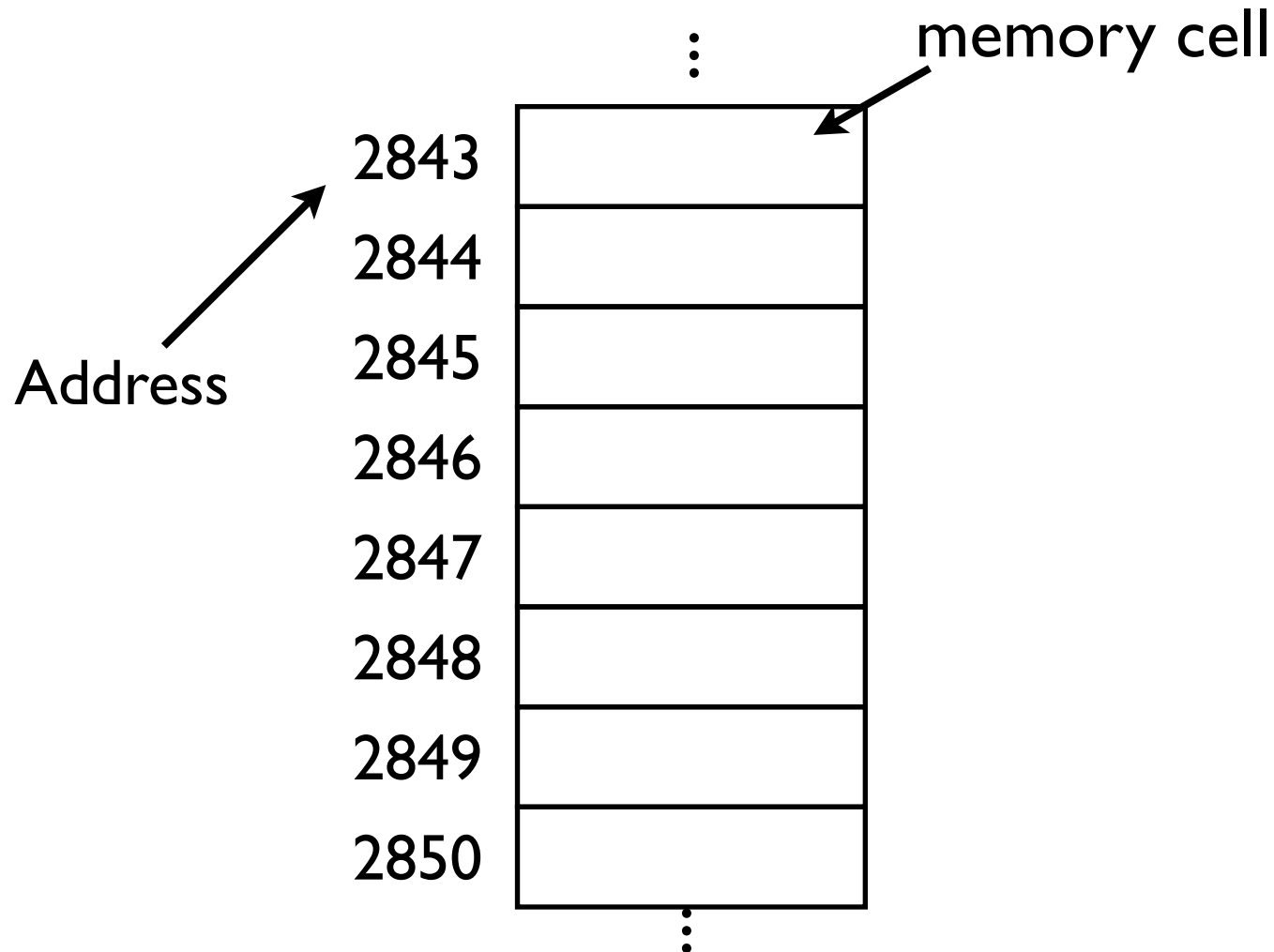
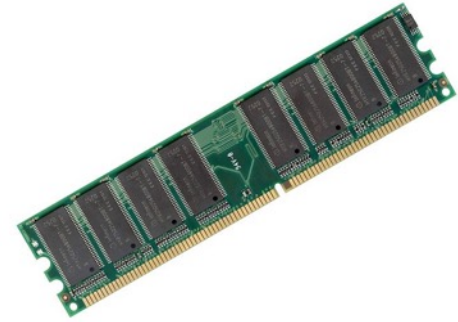
```
y = x
```

```
x[0] = 4
```

```
print(x, y)      [4, 2, 3] [4, 2, 3]
```


immutable vs mutable

Memory (Storage)
Closer look at RAM (Main memory)



immutable vs mutable

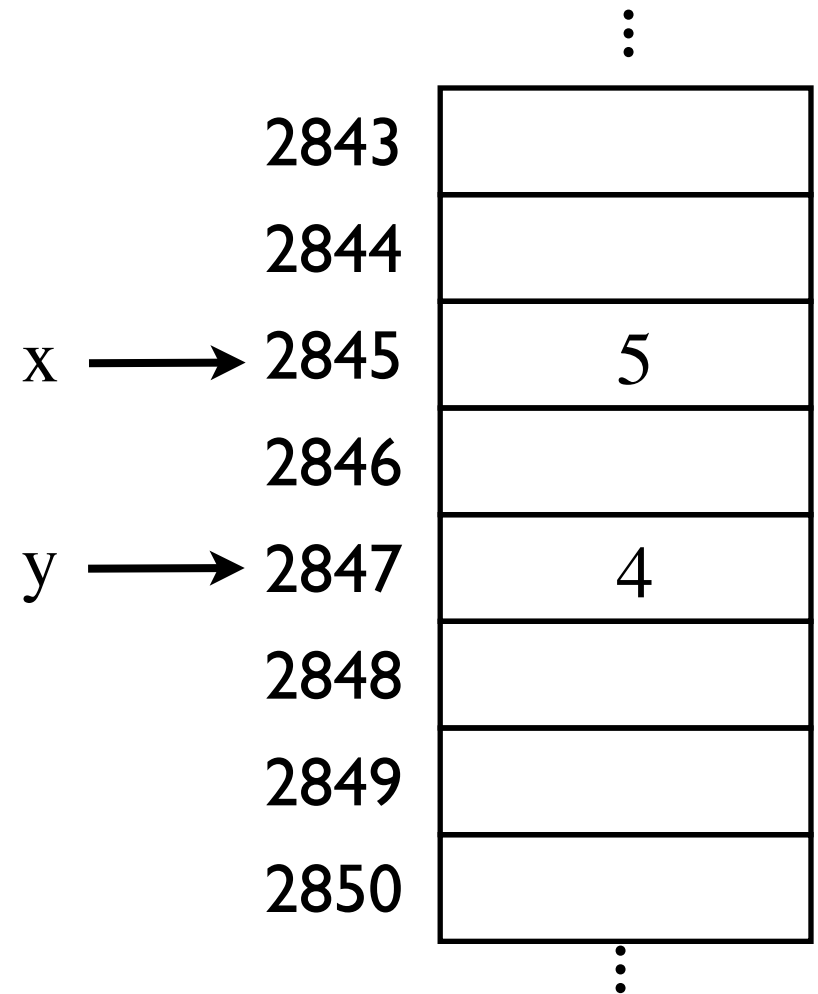
Immutable objects

x = 5

y = 4

x = 1

y = 2



immutable vs mutable

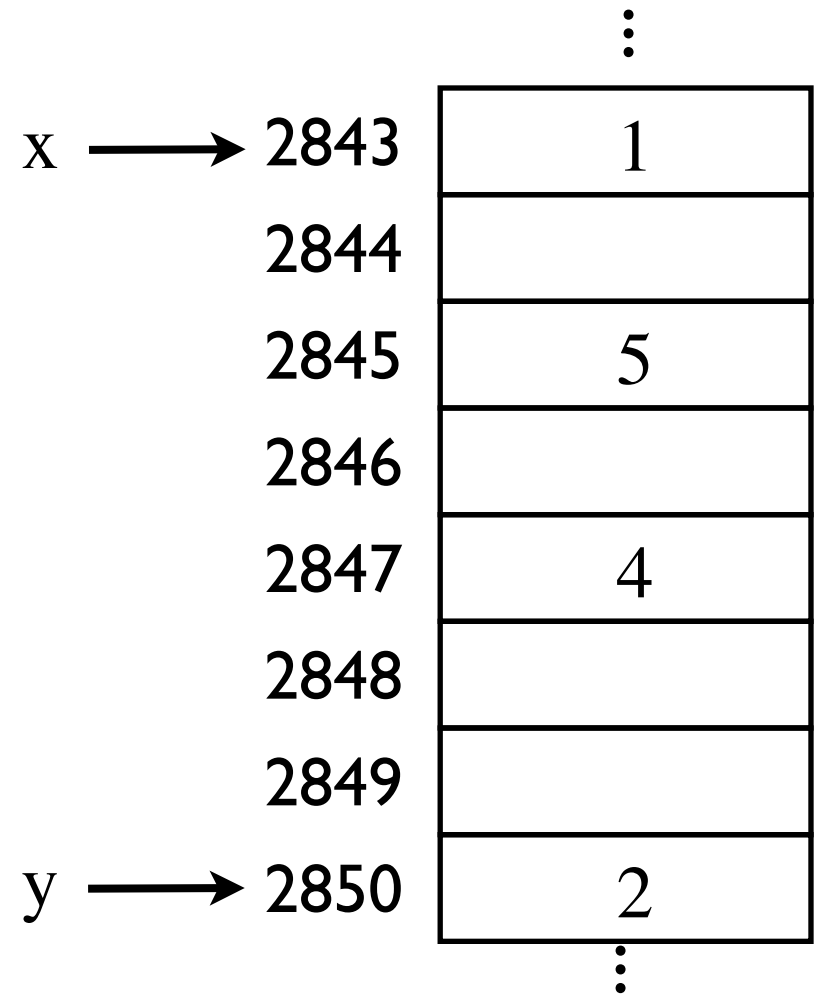
Immutable objects

x = 5

y = 4

x = 1

y = 2



immutable vs mutable

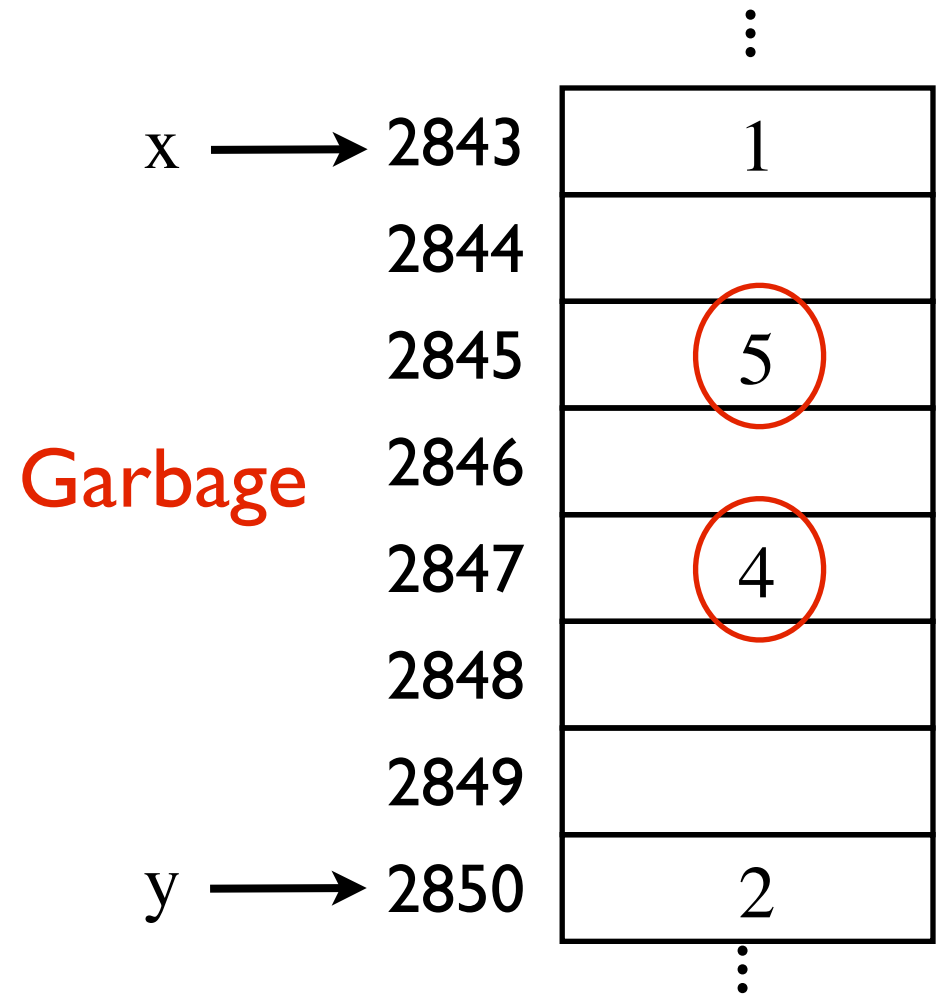
Immutable objects

x = 5

y = 4

x = 1

y = 2



immutable vs mutable

Immutable objects

x = 5

y = 4

x = 1

y = 2

“Garbage is collected”

x → 2843

2844

2845

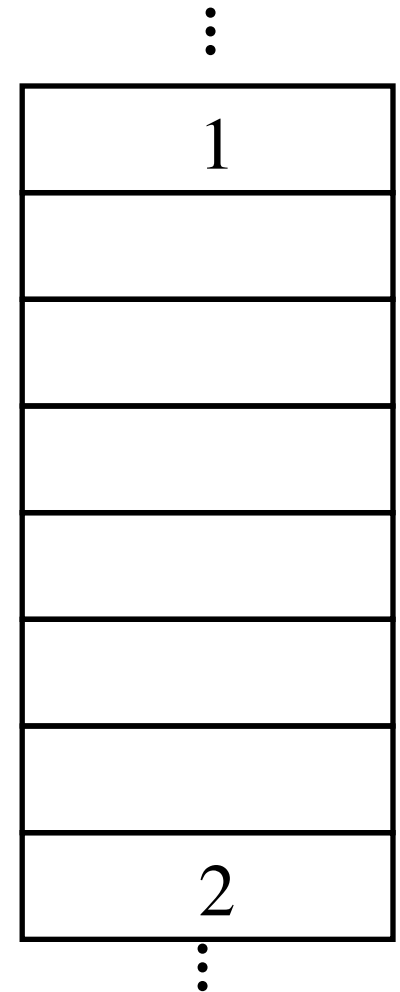
2846

2847

2848

2849

y → 2850



immutable vs mutable

Immutable objects

$x = 5$

$x \longrightarrow 5$

$y = 4$

$y \longrightarrow 4$

immutable vs mutable

Immutable objects

$x = 5$

$x \longrightarrow 5$

$y = 4$

$x = 1$

$y \longrightarrow 4$

immutable vs mutable

Immutable objects

$x = 5$

$x \longrightarrow 5$

$y = 4$

$x = 1$

$y \longrightarrow 4$

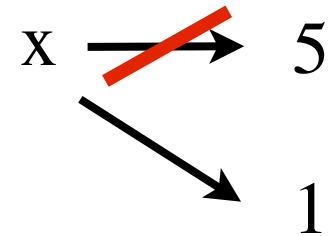
immutable vs mutable

Immutable objects

x = 5

y = 4

x = 1



immutable vs mutable

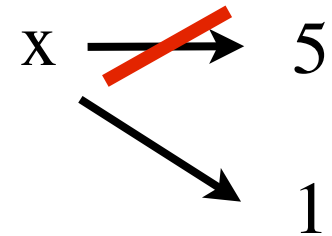
Immutable objects

x = 5

y = 4

x = 1

y = 2



immutable vs mutable

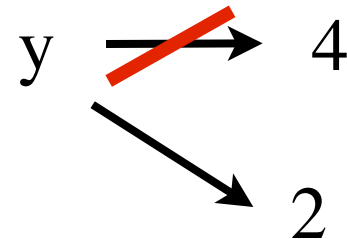
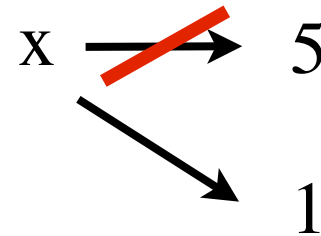
Immutable objects

x = 5

y = 4

x = 1

y = 2



immutable vs mutable

Immutable objects

$x = 5$

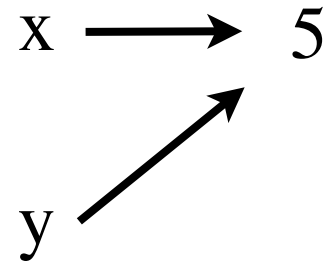
$x \longrightarrow 5$

immutable vs mutable

Immutable objects

$x = 5$

$y = x$



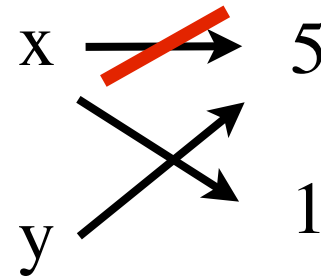
immutable vs mutable

Immutable objects

x = 5

y = x

x = 1



print(x, y) 1 5

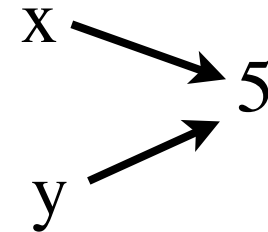
immutable vs mutable

Immutable objects

$x = 5$

$y = x$

In reality:



In practice:

$x \longrightarrow 5$

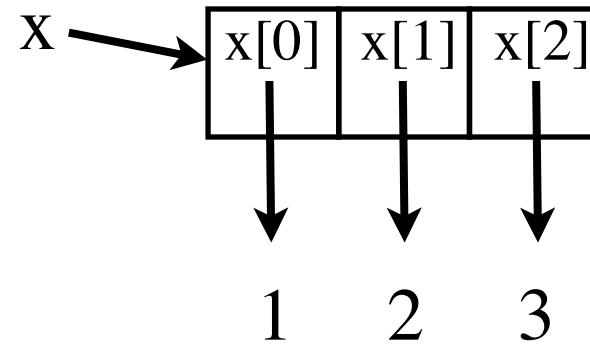
$y \longrightarrow 5$

(seems like a good thing)

immutable vs mutable

Mutable objects

`x = [1, 2, 3]`



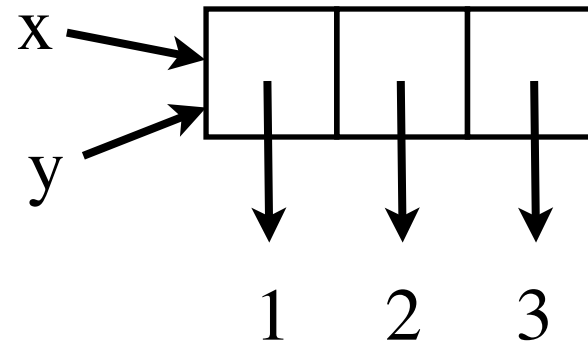
So actually,
a list is a sequence of references (variables)!

immutable vs mutable

Mutable objects

`x = [1, 2, 3]`

`y = x`



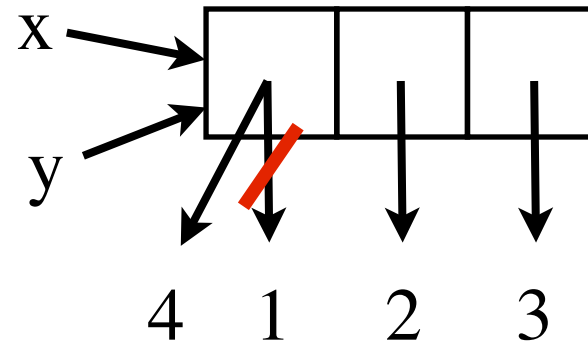
immutable vs mutable

Mutable objects

`x = [1, 2, 3]`

`y = x`

`x[0] = 4`



immutable vs mutable

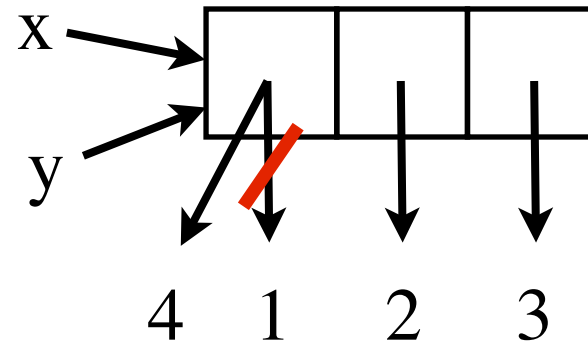
Mutable objects

```
x = [1, 2, 3]
```

```
y = x
```

```
x[0] = 4
```

```
print(y[0])    4
```



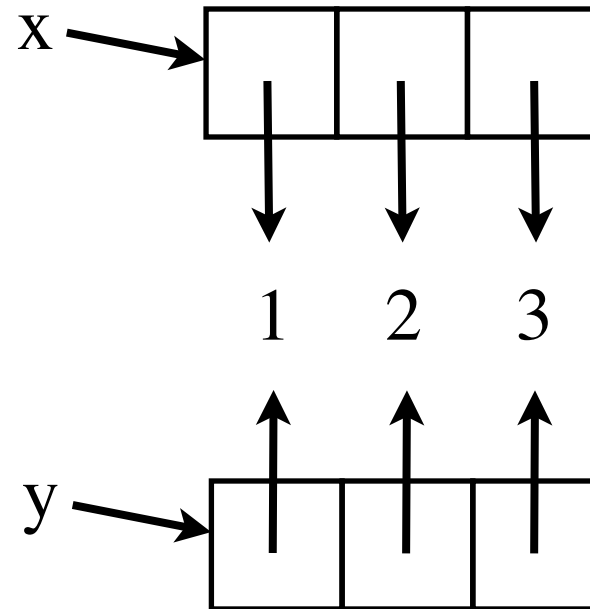
x and y are aliases.

immutable vs mutable

Mutable objects

```
x = [1, 2, 3]
```

```
y = [1, 2, 3]
```



```
print(x == y)    True
```

```
print(x is y)    False
```

```
print(x[0] is y[0]) True
```

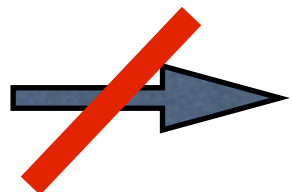
immutable vs mutable

With simpler data types, **immutability** is useful.
(no side effects)

With complex data types, **mutability** and **aliasing** is useful.
(avoid copying large data)

Suppose you have a list of names.

You add another name to the list



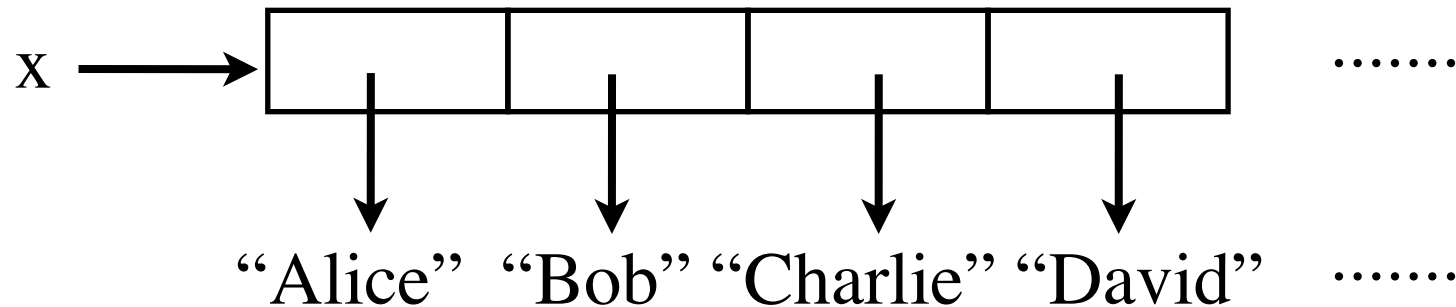
copy the whole list.

immutable vs mutable

If lists were immutable:

`x = ["Alice", "Bob", "Charlie", "David",]`

a million
names



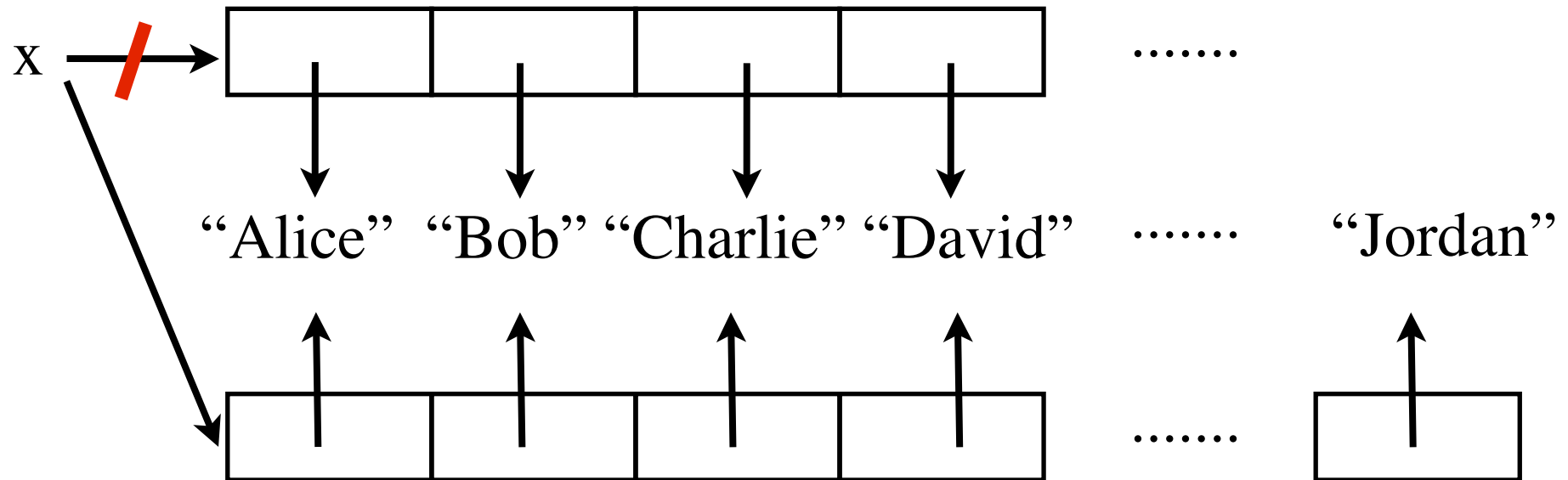
immutable vs mutable

If lists were immutable:

```
x = ["Alice", "Bob", "Charlie", "David", .....]
```

a million
names

```
x += ["Jordan"]
```



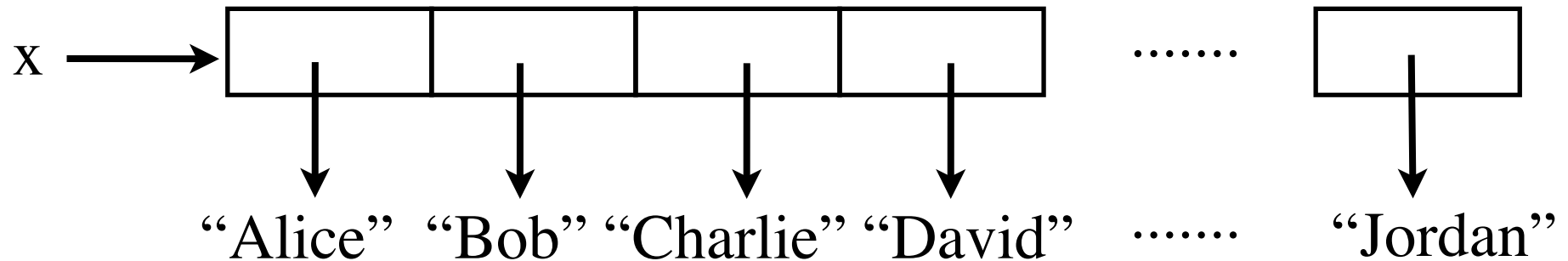
immutable vs mutable

But lists are mutable

```
x = ["Alice", "Bob", "Charlie", "David", .....]
```

a million
names

```
x += ["Jordan"]
```



immutable vs mutable

```
def square(x):  
    x = x**2  
    return x
```

```
n = 5
```

```
squaredNum = square(n)
```

```
print(n, squaredNum)
```

```
5 25
```

```
def square(a):  
    for i in range(len(a)):  
        a[i] = a[i]**2  
    return a    # Unnecessary
```

```
b = [1, 2, 3]
```

```
squaredList = square(b)
```

```
print(b, squaredList)
```

```
[1, 4, 9] [1, 4, 9]
```

Original b is destroyed

immutable vs mutable

```
def square(x):  
    x = x**2  
    return x
```

```
n = 5
```

```
squaredNum = square(n)
```

```
print(n, squaredNum)
```

```
5 25
```

import copy

```
def square(a):
```

```
    ➔ a = copy.copy(a)  
    for i in range(len(a)):  
        a[i] = a[i]**2  
    return a
```

```
b = [1, 2, 3]
```

```
squaredList = square(b)
```

```
print(b, squaredList)
```

```
[1, 2, 3] [1, 4, 9]
```

Original b is not destroyed

immutable vs mutable

Strings vs Lists

names = “Alice,Bob,Charlie,...” **a million
names**

Suppose you want to change Bob to William:

```
names = names.replace(“Bob”, “William”)
```

Creates a new string with a million names.

immutable vs mutable

Strings vs Lists

`names` = ["Alice", "Bob",] a million names

`changeName(names, "Bob", "William")`

```
def changeName(a, oldName, newName):  
    for index in range(len(a)):  
        if (a[index] == oldName):  
            a[index] = newName
```

`names` and `a` are aliases.
changes to `a` affect `names`.

The list of names is never duplicated/recreated.

immutable vs mutable

Strings vs Lists

Immutable ----> make copy every time you change it.

If dealing with huge strings, or
need to modify a string many times:

convert the string to a list first:

```
longText = list("Once upon a time, in a land far far away...")
```

converting the list back to a string:

```
longTextString = "".join(longText)
```

Digression: Alan Turing (1912-1954)



British mathematician, logician,
cryptanalyst, computer scientist.

Father of computer science and
artificial intelligence.

List operators and methods

2 types:

Destructive

- modifies original list

Non-destructive

- does not modify original list
- creates a new list

(with strings, for example, this is what happens)

List operators and methods

Adding elements

Destructive

```
a = [1, 2, 3]
a.append(4)
a = [1, 2, 3, 4]

a.extend([5, 6])
a = [1, 2, 3, 4, 5, 6]

a += [7, 8]  # same as extend
a = [1, 2, 3, 4, 5, 6, 7, 8]

a.insert(1, 1.5)
a = [1, 1.5, 2, 3, 4, 5, 6, 7, 8]
```

NonDestructive

```
a = [1, 2, 3]
b = a + [4]
b = [1, 2, 3, 4]  a = [1, 2, 3]

c = b + [5, 6]
c = [1, 2, 3, 4, 5, 6]
b = [1, 2, 3, 4]

d = c[:1] + [1.5] + c[1:]
d = [1, 1.5, 2, 3, 4, 5, 6]
```


IMPORTANT

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

```
b = a
```

```
a += [4]
```

```
print(a)    [1, 2, 3, 4]
```

```
print(b)    [1, 2, 3, 4]
```

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

```
b = a
```

```
a = a + [4]
```

```
print(a)    [1, 2, 3, 4]
```

```
print(b)    [1, 2, 3]
```

a += [4]

not same as

a = a + [4]

List operators and methods

Removing elements

Destructive

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

```
a.remove(3)
```

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

```
a.remove(3)
```

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

```
a.pop()
```

```
a = [1, 2, 1, 2, 1, 2]
```

```
print(a.pop(0)) 1
```

```
a = [2, 1, 2, 1, 2]
```

```
a[1:3] = [ ]
```

```
a = [2, 1, 2]
```

```
del a[1:]
```

```
a = [2]
```

NonDestructive

```
a = [2, 1, 2, 1, 2]
```

```
b = a[:1] + a[3:]
```

```
b = [2, 1, 2] a = [2, 1, 2, 1, 2]
```

List operators and methods

Common Mistakes

```
def remove(someList, element):  
    for index in range(len(someList)):  
        if (someList[index] == element):  
            someList.pop(index)
```

Index range changes
every time you pop.

```
def total(someList):  
    t = 0  
    while(someList != []):  
        t += someList.pop()  
    return t
```

Never change the list
if you don't need to!

```
a = [1, 2, 3, 1, 2, 3, 1, 2, 3]  
print(total(a))  
print(a)      []
```

List operators and methods

sort vs sorted

Destructive

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

```
a.sort()
```

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

NonDestructive

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

```
b = sorted(a)
```

```
b = [1, 1, 2, 2, 3, 3]
```

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

List operators and methods

finding an element

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

```
print(a.index(2))    1
```

```
print(a.find(2))     ERROR: no method called 'find'
```

```
print(a.index(4))     ERROR: 4 is not in the list
```

```
if (4 in a):  
    print("4 is at index", a.index(4))  
else:  
    print("4 is not in the list.")
```

List operators and methods

others

<https://docs.python.org/3/library/stdtypes.html#typeseq-mutable>

<https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>

Summary

Destructive
(modifies the given list)

`+=`

every method that
manipulates the list

`del` statement

NonDestructive

`+`, `*`

functions

slicing

Be careful about aliasing
(especially with function parameters)

Swapping

Incorrect:

x = 5

y = 0

swap x and y

x = y

y = x

Correct:

x = 5

y = 0

swap x and y

temp = x

x = y

y = temp

x = 5

y = 0

swap x and y

x, y = y, x

a = [1, 2, 3]

swap a[0] and a[2]

a[0], a[2] = a[2], a[0]

List comprehension

A concise way to create lists.

[<expr> <for clause> (additional/optional for and if clauses)]

```
a = [x for x in range(10)]
```

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Same as:

```
a = []
```

```
for x in range(10):  
    a.append(x)
```

Could of course just do this instead:

```
a = list(range(10))
```

List comprehension

A concise way to create lists.

[<expr> <for clause> (additional/optional for and if clauses)]

```
squares = []  
for x in range(10):  
    squares.append(x**2)
```

```
squares = [0, 1, 4, 9, 25, 36, 49, 64, 81]
```

```
squares = [x**2 for x in range(10)]
```

```
squares = [0, 1, 4, 9, 25, 36, 49, 64, 81]
```

```
primeSquares = [x**2 for x in range(10) if isPrime(x)]
```

```
primeSquares = [4, 9, 25, 49]
```

Tuples

The immutable brother of lists

Tuples

```
myTuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
myTuple = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10  # not recommended
```

```
myTuple = (1, "hello", 3.14, True)
```

```
myTuple = (1,)  # Put comma for one element tuple
```

```
myTuple[0] = 2  ERROR
```

parallel assignments

```
x, y = 1, 2
```

Tuples

return multiple values in a function

```
def firstPrimeInList(a):  
    for i in range(len(a)):  
        if (isPrime(a[i])):  
            return (i, a[i])  
return -1
```

Before recitation tomorrow:

Go over list notes

Read the quiz-practice problems

Make use of office hours