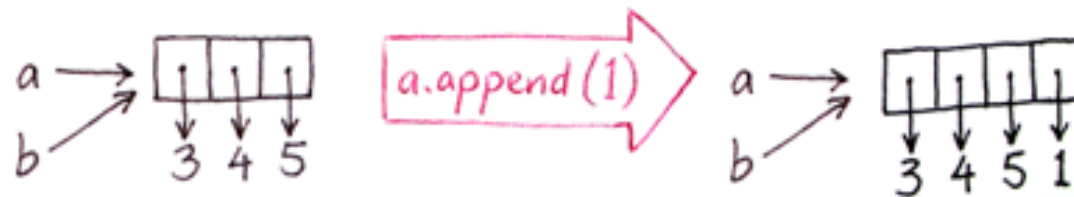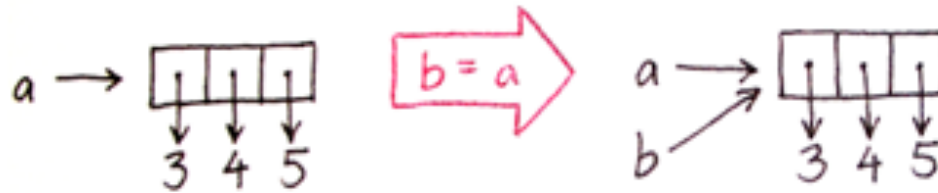# 15-112
Fundamentals of Programming

Week 2 - Lecture 3:
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 = "hw2-1 was hard"

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

```python
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])          [1, 2, 3, 4, 5, 6]
a = [0] * 5
print(a)                              [0, 0, 0, 0, 0]

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

Lists: interesting example

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



x = [1, 2, 3]
y = x
x[0] = 4
print(x, y)          [4, 2, 3]  [4, 2, 3]
```
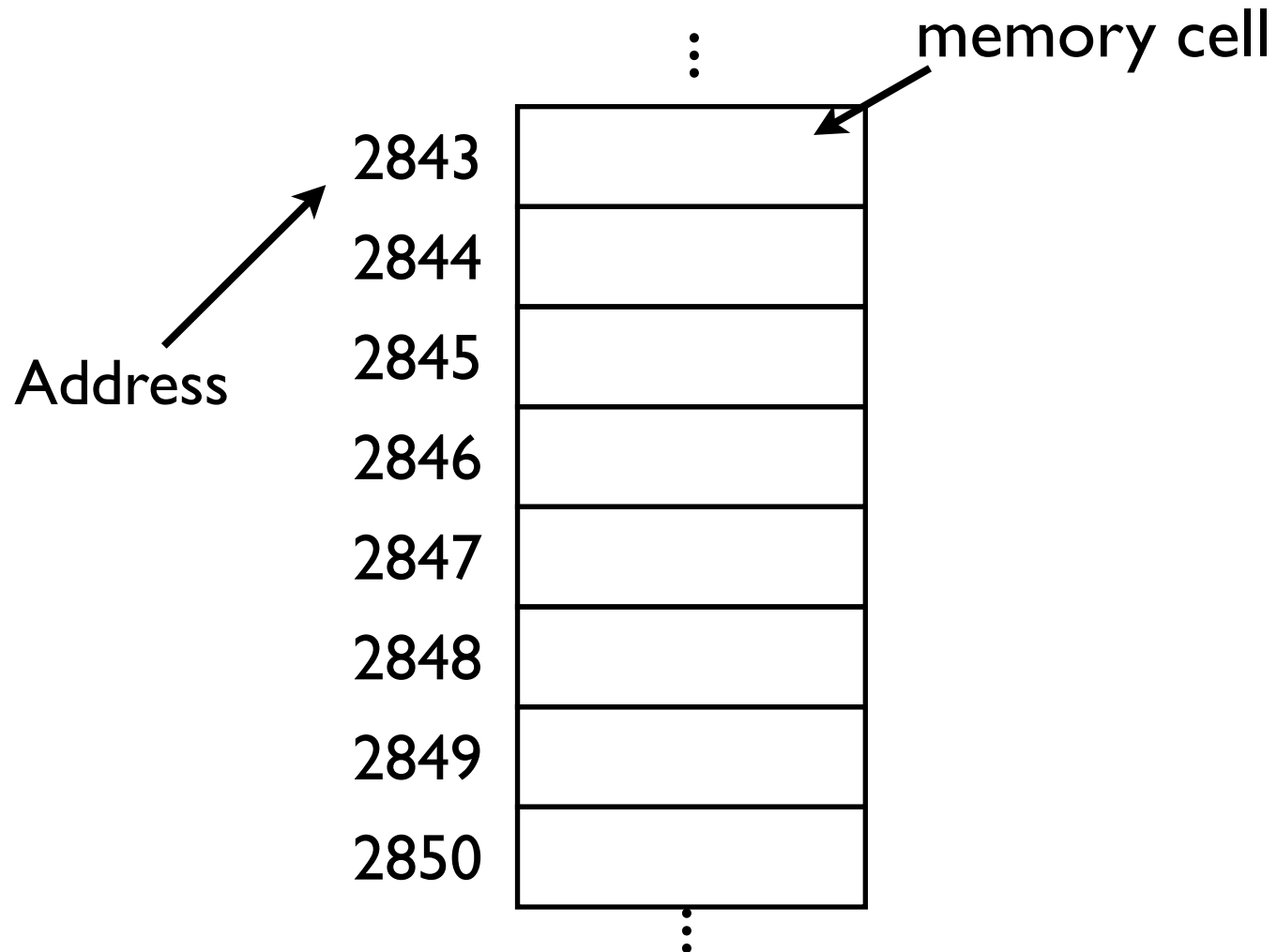
immutable vs mutable

Let's take a closer look

memory cell

⋮

Address	
2843	
2844	
2845	
2846	
2847	
2848	
2849	
2850	

⋮

Immutable objects

x = 5

y = 4

x = 1

y -= 2

⋮

2843

2844

x ⟶ 2845   5

2846

y ⟶ 2847   4

2848

2849

2850

⋮
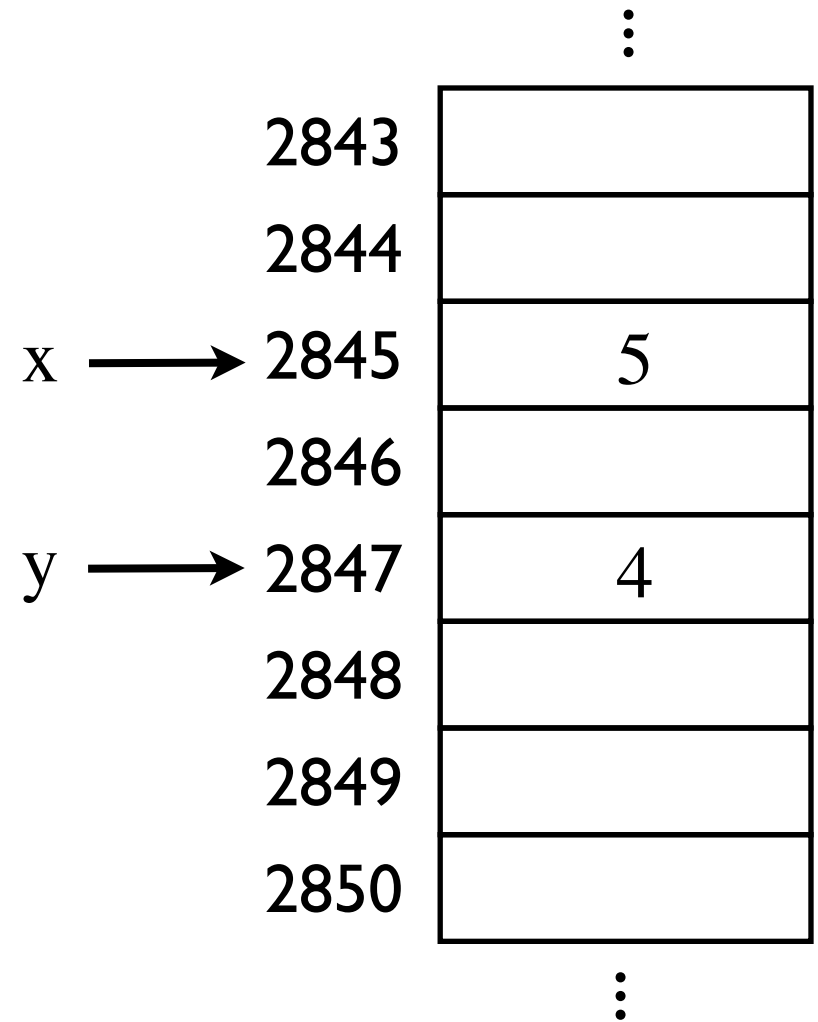
Immutable objects

x = 5

y = 4

x = 1

y -= 2

x ⟶ 2843

2844

2845

2846

y ⟶ 2847

2848

2849

2850

	1
	5
	4

Immutable objects

x = 5

y = 4

x = 1

y -= 2

$x \longrightarrow$ 2843   1

2844

2845   5

2846

2847   4

2848

2849

$y \longrightarrow$ 2850   2

Immutable objects

$x = 5$

$y = 4$

$x = 1$

$y \mathrel{-}= 2$

Garbage

x → 2843	1	
2844		
2845	5	
2846		
2847	4	
2848		
2849		
y → 2850	2	

Immutable objects

x = 5

y = 4

x = 1

y -= 2

"Garbage is collected"

x ⟶ 2843

2844

2845

2846

2847

2848

2849

y ⟶ 2850

⋮

1
2

⋮

Immutable objects

x = 5

y = 4

x ⟶ 5

y ⟶ 4

Immutable objects

x = 5

y = 4

x = 1

x $\longrightarrow$ 5

y $\longrightarrow$ 4

Immutable objects

x = 5

y = 4

x = 1

x ⟶̷ 5

↘ 1

y ⟶ 4

Immutable objects

x = 5

y = 4

x = 1

y -= 2

x ⟶̶ 5

x ⟶ 1

y ⟶ 4
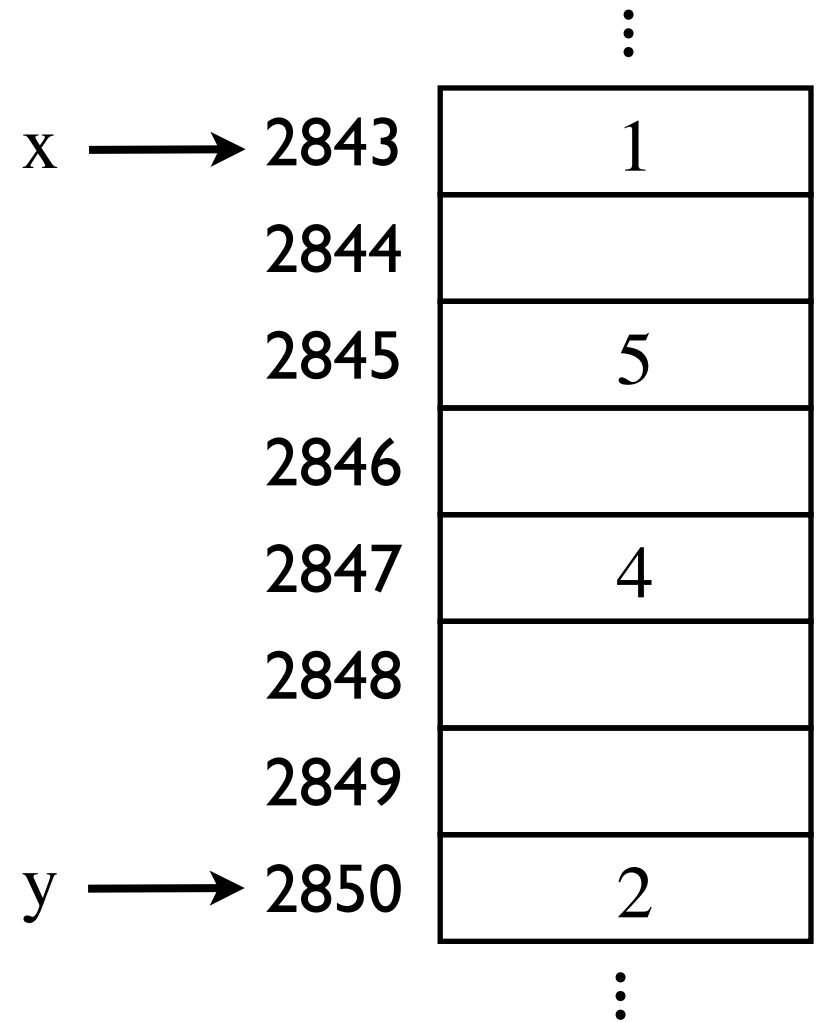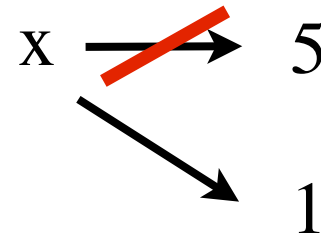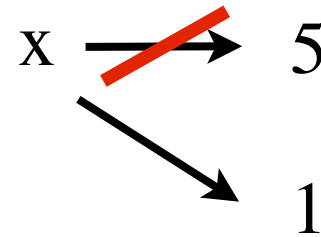
Immutable objects

x = 5

y = 4

x = 1

y -= 2
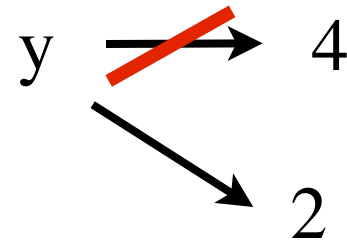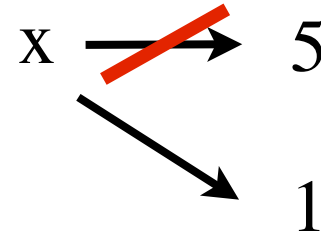
Immutable objects

x = 5

$$x \longrightarrow 5$$

Immutable objects
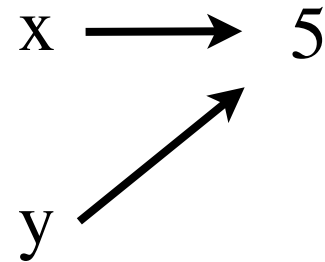
x = 5

y = x

x ⟶ 5

y ⟶ 5

Immutable objects

x = 5

y = x

x += 1

print(x, y)

x $\longrightarrow$ 5

y

Immutable objects

x = 5

y = x

x += 1

print(x, y)   6 5

Immutable objects

x = 5

y = x
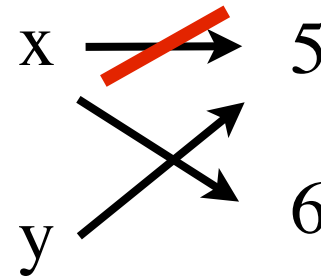
In reality:

$$x \searrow$$
$$\qquad 5$$
$$y \nearrow$$

In practice:

$$x \longrightarrow 5$$
$$y \longrightarrow 5$$
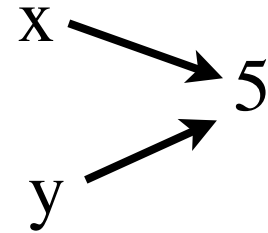
(seems like a good thing)

immutable vs mutable

Mutable objects

x = [1, 2, 3]

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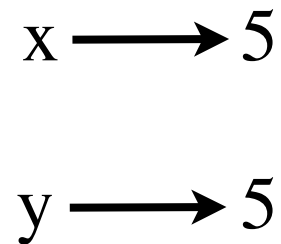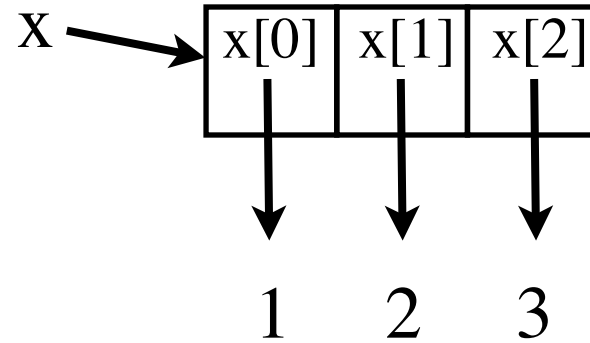a list is a sequence of references (variables)!

Mutable objects

x = [1, 2, 3]

y = x

Mutable objects

x = [1, 2, 3]

y = x

x[0] = 4

Mutable objects

x = [1, 2, 3]

y = x

x[0] = 4

print(y[0])   4

x and y are **aliases**.

Mutable objects

x = [1, 2, 3]

y = [1, 2, 3]

x → ▭▭▭

1 2 3

y → ▭▭▭

1 2 3
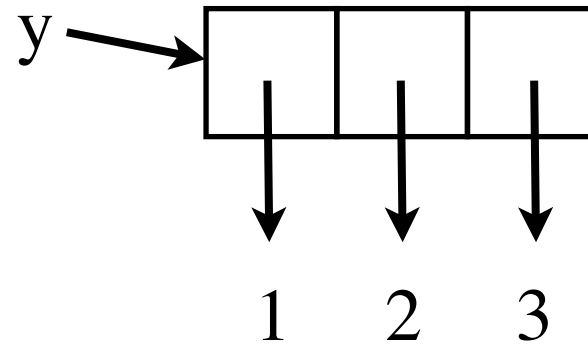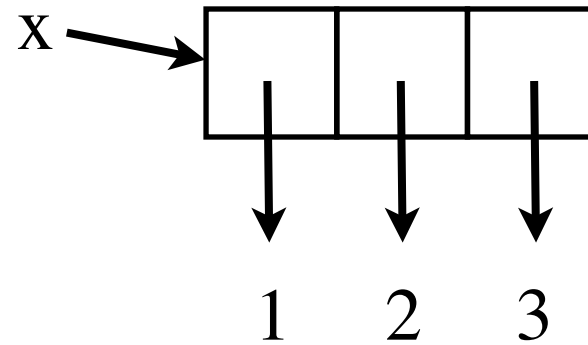
print(x == y)   True

print(x **is** y)   False

immutable vs mutable

With simpler data types, immutabality 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
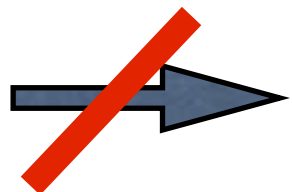
copy the whole list.

immutable vs mutable

If lists were __immutable__:

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

a million names

If lists were __immutable__:

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

x += ["Jordan"]
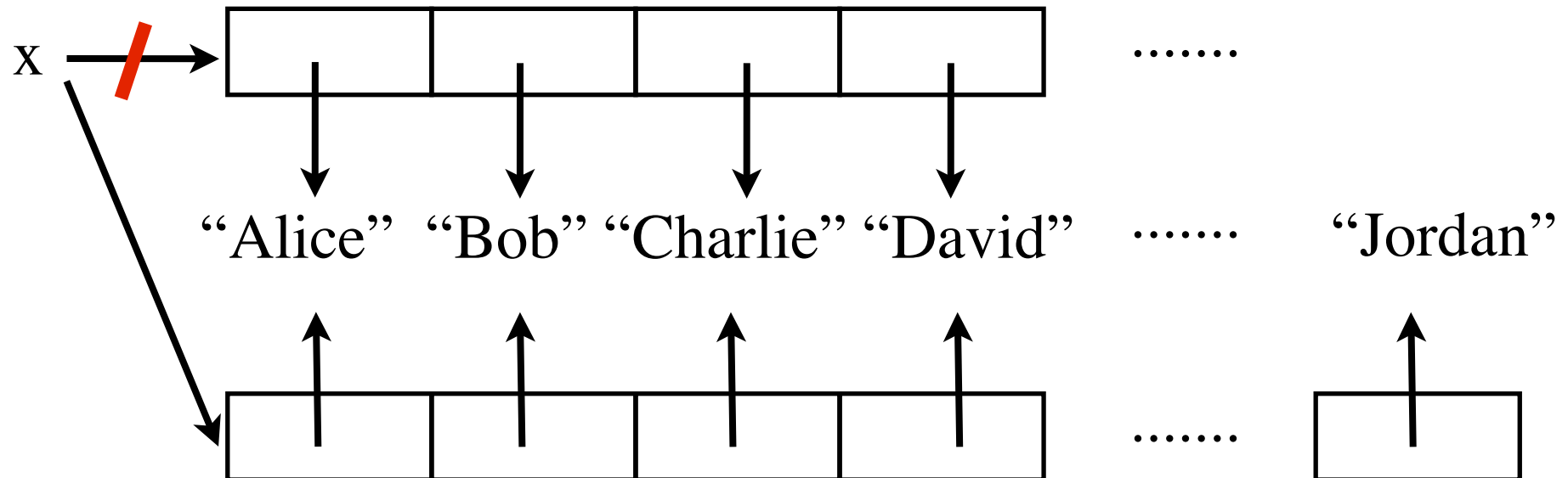
a million names

immutable vs mutable

But lists are **mutable**

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

x += ["Jordan"]
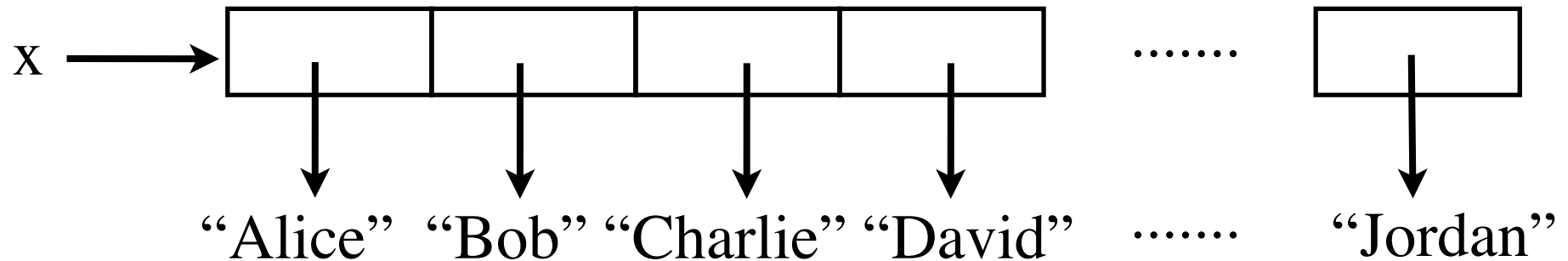
a million names

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)

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	NonDestructive

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]   a = [1, 2, 3]
b = a   b = a
a += [4]   a = a + [4]
print(a)   [1, 2, 3, 4]   print(a)   [1, 2, 3, 4]
print(b)   [1, 2, 3, 4]   print(b)   [1, 2, 3]

a += [4]   **not same as**   a = a + [4]

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

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

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

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.")

others

https://docs.python.org/3/library/stdtypes.html#typesseq-mutable

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

Summary

Destructive (modifies the given list)	NonDestructive
+=	+, *
every method that manipulates the list	functions
del statement	slicing

Be careful about aliasing
(especially with function parameters)

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
```

Exercise Problem

Lockers Problem

1   2   3   4   5   6   7   n