References and Memory

15-110 – Monday 09/26

Announcements

- Hw2 was due today
  - How did it go?

Learning Goals

- Recognize whether two values have the same **reference** in **memory**
- Recognize the difference between **destructive** vs. **non-destructive** functions/operations on **mutable** data types
- Use **aliasing** to write functions that destructively change lists

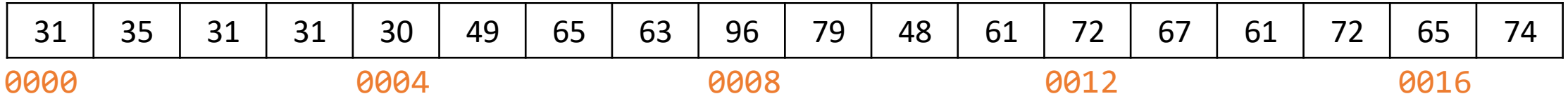
References and Memory

Computer Memory Holds Data

Recall from the Data Representation lecture that all data on your computer is represented as **bits** (0s and 1s).

Your computer's memory is a very long sequence of bytes (8 bits), which are interpreted with different abstractions to become different types. Each byte has its own **address**.

When you write a Python program, every variable you create is associated with a different segment of memory. The way variables connect to memory becomes more complicated when we use data structures.

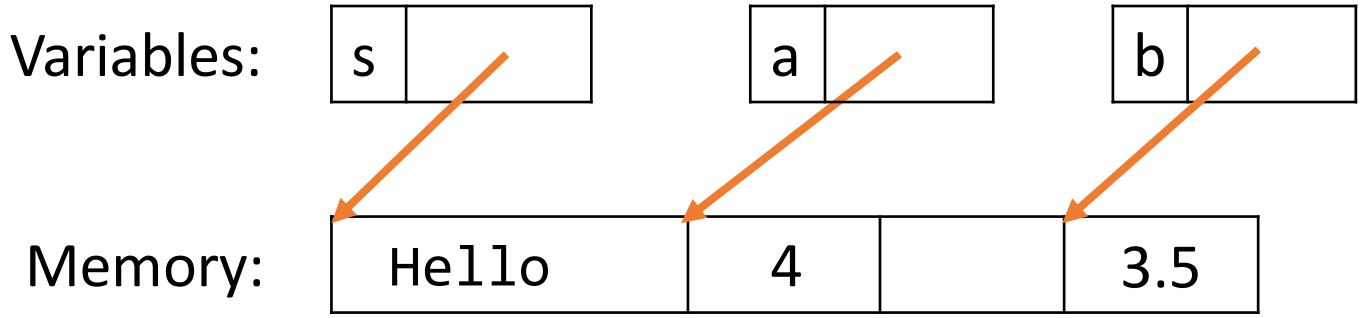


References are Memory Addresses

A **reference** (often called a pointer) is a specific address in memory. References are used to connect variables to their values.

When we set a variable equal to a value, we keep the variable and value **one step apart**. The variable only has access to a reference, which points to the value. If Python goes to the reference's address, it can retrieve the value stored there.

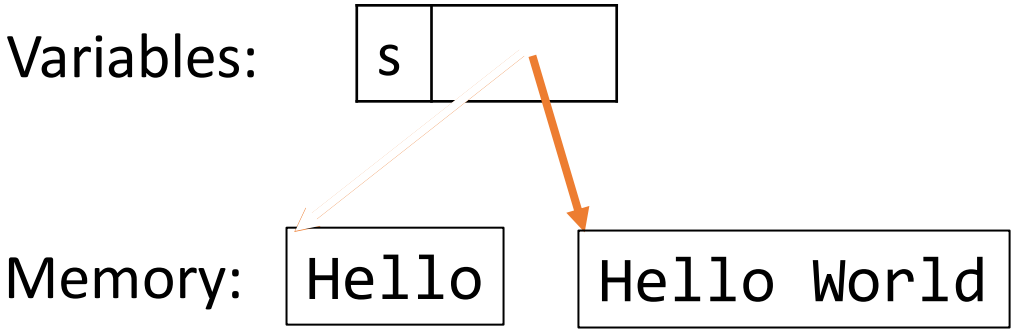
```
s = "Hello"
a = 4
b = 3.5
```



Updating a Variable Changes the Reference

When we set a variable to a new value, Python makes a **new data value** and reassigns the variable to reference the new value. It does not change the old value in memory at all.

```
s = "Hello"
s = s + " World"
```



Analogy: Lockers and Nametags

You can think of Python's memory as a series of lockers, each with its own number. The item inside a locker is the data value it holds.

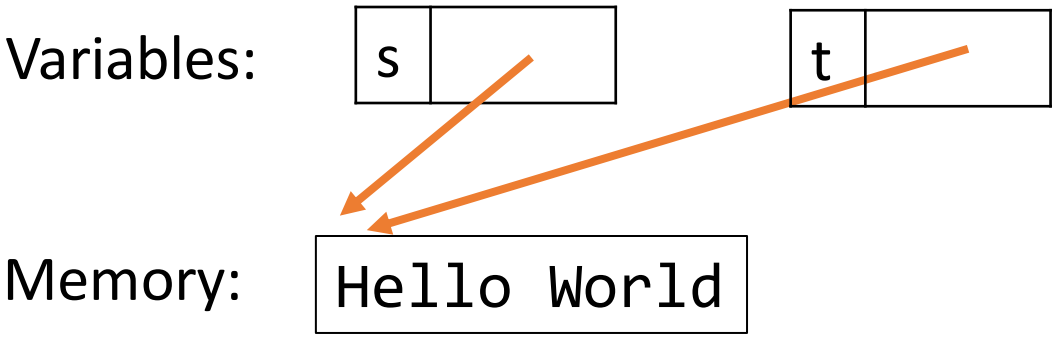
A variable is then a nametag sticker. When you stick a nametag onto a locker, it 'points to' the item in that locker. If you move the nametag onto a different locker, the original locker's contents don't change.

Copying a Variable Copies the Reference

What happens when we set a new variable equal to an old one? We don't need to create a new data value in a new memory address; Python just **copies the reference** instead.

This is like taking a new nametag and putting it on the same locker as another nametag.

```
s = "Hello World"
t = s
```

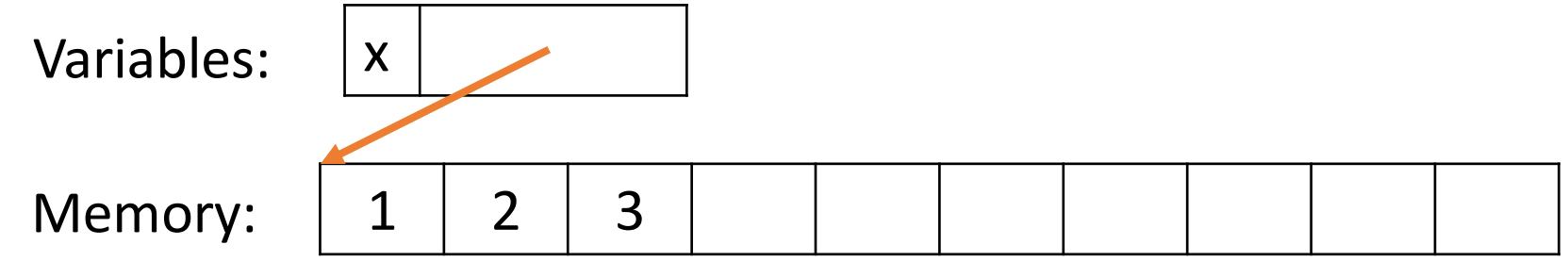


Lists Take Up Adjacent Addresses

When we set a variable to a **list** (or another data structure), Python sets aside a large place in memory for the data values it will hold.

By breaking up that large chunk of memory into parts, Python can assign each value in the list a location, ordered sequentially.

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



Technically each index also holds a reference to a new location, but that's out of scope for this course

Analogy: A List is a Locker With Shelves

You can think of the list memory as a single locker (the starting reference) broken up with several shelves.

Each shelf can hold its own item (data value) and has its own reference.

This allows us to change memory in new and interesting ways.

Mutable vs Immutable Values

List Values Can Be Changed

Because of how lists are stored in memory, the values in a list can be **changed directly** without reassigning the variable.

We can change a list by setting a list index to a new value, like how we would set a variable to a value.

```
lst = [ "a", "b", "c" ]
lst[1] = "foo"
lst # [ "a", "foo", "c" ]
```

Some List Methods Change the List

We can also modify a list directly, to add or remove elements from it, using some **list methods**. These methods change the list **without using variable assignment at all**.

```
lst = [ 1, 2, "a" ]
lst.append("b") # adds the element to the end of the list
lst # [ 1, 2, "a", "b" ]
```

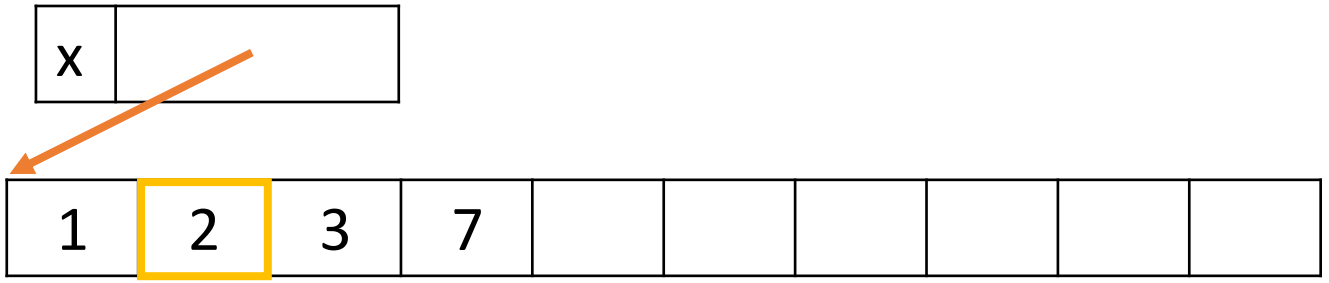
Note that we do not set `lst = lst.append`; the list is changed **in place**. In fact, the `append` method returns `None`, not a list.

Modifying Lists in Memory

How do these methods work? The large space set aside for the list values allows Python to add and remove values from the list **without running out of room in memory**. It's like having tons of empty shelves in the locker and putting the item on one of them.

This makes it easy (and fast!) to locate a specific value based on its index.

```
x = [1, 2, 3]
x.append(7)
print(x[1])
```



Lists are Mutable; Strings are Immutable

We call data types that can be modified without reassignment this way **mutable**. Data types that cannot be modified directly are called **immutable**.

All the other data types we've learned about so far – integers, floats, Booleans, and strings – are immutable. In fact, if we try to set a string index to a new character, we'll get an error. We have to set the entire variable equal to a new value if we want to change the string.

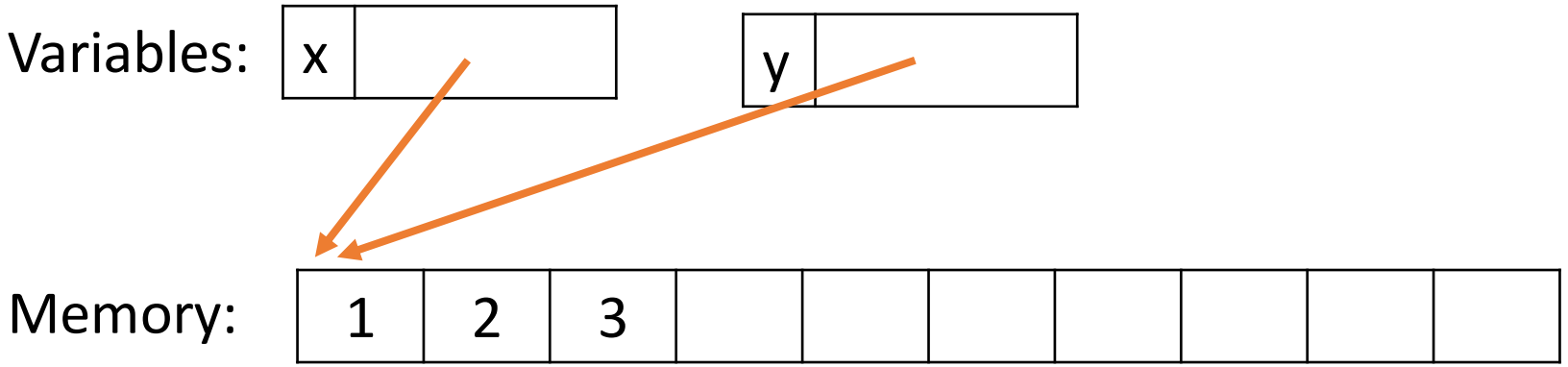
```
s = "abc"
s[1] = "z" # TypeError
s = s[:1] + "z" + s[2:]
```

Copying Lists in Memory

We showed before that when we copy a variable into a new variable, the **reference** is copied, not the value.

This is true for lists as well; an example is shown below.

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



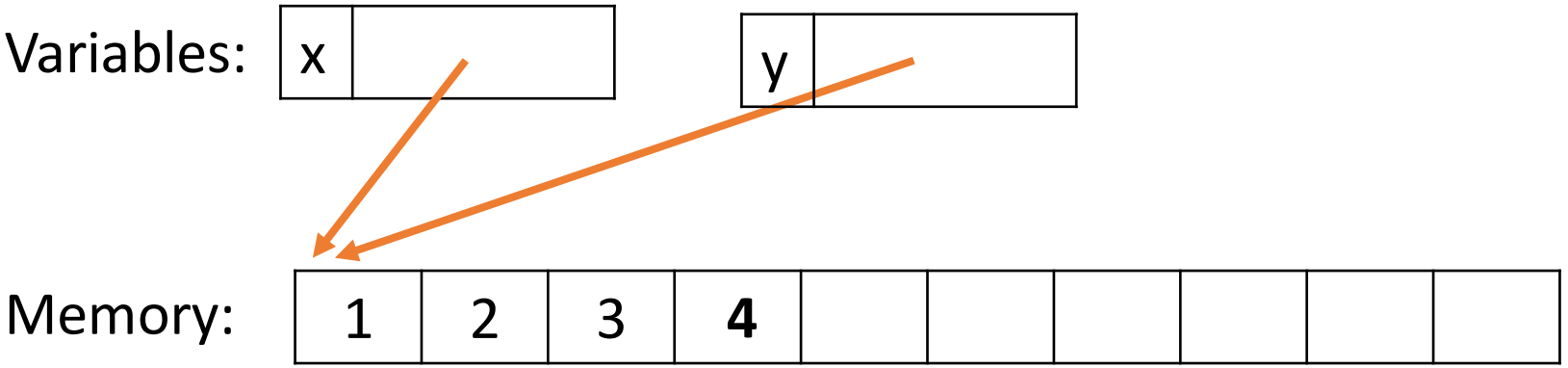
You do: what happens to the values in `x` and `y` if we add the line `y.append(4)` to the end of this code snippet?

Reference-Sharing Lists Share Changes

When a direct action is done on a list, that action affects the **data values,** not the variable. Any lists that share a reference with the original list will see the same changes!

We call lists that share a reference this way **aliased**.

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

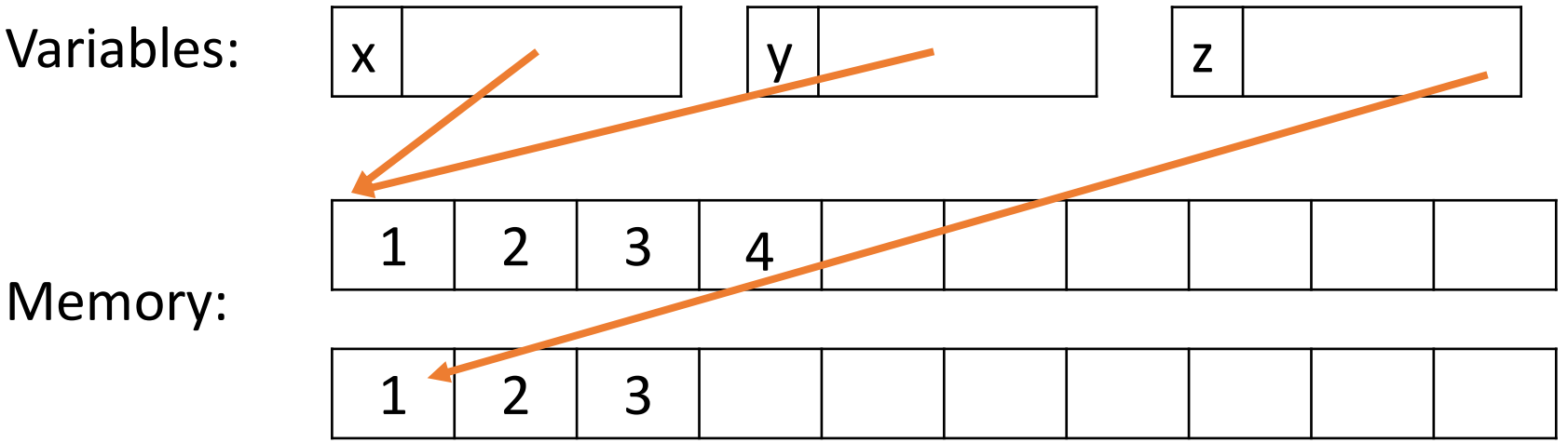


Copying Variables vs. Copying Values

Two variables won't be aliased just because they contain the same values. Their references need to point to the same place for them to be aliased.

In the following example, the lack of a **reference copy** keeps the list z from being aliased to x and y.

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]
x.append(4)
```



Break an Alias with List Concatenation

If you have two variables that are aliased and you don't want them to be aliased, you need to 'break' the alias between them. This is done by setting one of the variables equal to a new data value with the same values as the original list.

The easiest way to do this is to concatenate the empty list to the original list. Python doesn't recognize that the second list is empty, so it will create an entirely new list in memory.

```
a = ["A", "B", "C"]
b = a # a and b are aliased
a = a + [ ] # a now has a new reference, but the same values
```

Why does this work? Variable assignment with list concatenation is **non-destructive**.

Destructive vs. Non-destructive

Two Ways of Modifying Lists

Whenever we want to modify a list (by changing a value, adding a value, or removing a value), we can choose to do so **destructively** or **non-destructively**.

Destructive approaches change the data values without changing the variable reference. Any aliases of the variable will see the change as well, since they refer to the same list.

Non-destructive approaches make a new list, giving it a **new reference**. This 'breaks' the alias and doesn't change the previously-aliased variables.

Destructive Methods are Efficient

Why would we ever want to use a destructive approach instead of a simpler non-destructive approach?

Destructive approaches are **more efficient**. Instead of needing to copy all the values into a new place in memory, you only change a small part of the existing memory. This saves time and space in memory.

Real-life example: maybe you use a list to track all the patients in a hospital. The list is quite long, and new patients are constantly being admitted (added) and discharged (removed). Modifying the list destructively is much faster than rewriting it every time a change is made!

Two Ways to Add Values

How do we add a value to a list **destructively**? Use destructive methods - `append`, `insert`, or `extend`.

```
lst = ["A", "B", "C"]
lst.append("E") # add value to the end
lst.insert(0, "foo") # inserts 2nd param into 1st param index
lst.extend(["F", "G"]) # adds multiple elements
```

How do we add a value to a list **non-destructively**? Use variable assignment with list concatenation.

```
lst = ["A", "B", "C"]
lst = lst + ["E"] # note that "E" needs to be in its own list
# warning: 'lst +=' and 'lst = lst +' behave differently!
lst = lst[:len(lst)//2] + ["F"] + lst[len(lst)//2:]
```

Two Ways to Remove Values

How do we remove a value from a list **destructively**? Use destructive methods - `remove` or `pop`.

```
lst = ["A", "B", "C"]
lst.remove("A") # removes the given element from the list once
lst.pop(1) # removes the element at given index from the list
```

How do we remove a value from a list **non-destructively**? Use variable assignment with list slicing.

```
lst = ["A", "B", "C"]
lst = lst[1:]
lst = lst[:len(lst)-1]
```

Other Destructive Methods

There are a few other destructive methods that may come in handy:

```
lst = [4, 1, 9, 2]
lst.sort() # sorts the list destructively

import random
random.shuffle(lst) # mixes the elements up destructively
```

And of course, list index assignment is also destructive!

```
lst[3] = 42 # changes the element at index 3 destructively
```

Activity: Which Lists are Aliased?

At the end of this set of operations, which lists will be aliased? What values will each variable hold?

```
a = [ 1, 2, "x", "y" ]
b = a
c = [ 1, 2, "x", "y" ]
d = c
a.pop(2)
b = b + [ "woah" ]
c[0] = 42
d.insert(3, "yowza")
```

Writing Destructive Functions

Function Arguments/Parameters are Aliased

When you call a function with a mutable value as one of the arguments, that argument is **aliased** to the function's parameter variable. The same reference is used for the original argument and the parameter that the function uses.

This means that we can write our own functions that behave **destructively**, changing the data values in the given list directly instead of making a new list. This is valuable when we work with large datasets, as we don't want to make a copy of all the values each time we make a change.

```
def foo(lst):
    lst[1] = "bar"

x = [1, 2, 3]
print(foo(x)) # when lst is created, it copies x's reference
print(x) # now 2 has been replaced with "bar"
```

Destructive Functions Use Mutable Methods

When writing a destructive function, use index assignment and the mutable methods (`append`, `insert`, `extend`, `pop`, and `remove`) on the **parameter list** to change it as needed.

For example, the following code **destructively** doubles all the values in the given list of integers. Note that the function need not return `lst` because the parameter `lst` and the argument `x` **refer to the same values**. We usually have destructive functions return `None` as an indicator that they're destructive.

```
def destructiveDouble(lst):
    for i in range(len(lst)):
        lst[i] = lst[i] * 2

x = [1, 2, 3]
destructiveDouble(x)
print(x)
```

Non-Destructive Functions Make New Lists

If you want to make a function that is not destructive, you should instead set up a new list and fill it with the appropriate values. To be non-destructive, the parameters must **not** be changed.

The following code **non-destructively** creates a new list of all the doubles of values in the original list. This function **does** need to return the result, as the parameter is not changed. After the call to the function, the variable `x` will not have changed; `y` refers to the new list with all the values doubled.

```
def nonDestructiveDouble(lst):
    result = [ ]
    for i in range(len(lst)):
        result.append(lst[i] * 2)
    return result

x = [1, 2, 3]
y = nonDestructiveDouble(x)
print(x, y)
```

Sidebar: Destructive Looping

Looping with List Destruction

Looping over a list while changing it destructively can cause some odd side-effects because of how **loop control variables** are managed.

You'll often want to do destructive looping, so here are two tips for how to manage it.

`for` vs. `while`

It is **a very bad idea** to destructively add or remove elements in a list while looping over it with a **`for` loop**.

This will lead to unexpected and bad behavior. Why? The `range` is only calculated once.

```
lst = ["a", "a", "c", "d", "e"]
for i in range(len(lst)):
    if lst[i] == "a" or \
       lst[i] == "e":
        lst.pop(i)
```

Instead, use a **`while` loop** if you're planning to destructively change the list length inside a loop.

The list length is reevaluated when the while condition is checked each iteration.

```
lst = ["a", "a", "c", "d", "e"]
i = 0
while i < len(lst):
    if lst[i] == "a" or \
       lst[i] == "e":
        lst.pop(i)
    else:
        i = i + 1
```

`break` to exit early

What if you want to destructively remove exactly one element from a list, then exit the loop immediately before you remove any others?

It's possible to design a loop control variable to do this, but it's often easier to use the `break` statement instead. As soon as the code reaches a `break`, it immediately exits the loop. (If loops are nested, it only exits the innermost loop).

```
lst = ["a", "a", "c", "d", "e"]
for i in range(len(lst)):
    if lst[i] == "a":
        lst.pop(i)
        break # exits immediately, only removes one "a"
```

Learning Goals

- Recognize whether two values have the same **reference** in **memory**
- Recognize the difference between **destructive** vs. **non-destructive** functions/operations on **mutable** data types
- Use **aliasing** to write functions that destructively change lists