Aliasing and Mutability

15-110 – Wednesday 02/19

Learning Goals

- Recognize how **aliasing** impacts the values held in mutable variables

- Recognize the difference between functions on mutable values that are **destructive** vs. **non-destructive**

- Use **2D lists** when reading and writing code to work on data over multiple dimensions

Last Time…

List Values Can Be Changed

Unlike the previous types we've worked with, the values in a list can be **changed directly**, without creating a new list that needs to be assigned to 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"
print(lst) # [ "a", "foo", "c" ]
```

Lists are Mutable; Strings are Immutable

We call data types that can be modified after being assigned **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:]
```

List Methods That Mutate

There are a set of list methods we can use that mutate (change) the list.

For each of these methods, we don't need to set the result to the variable- when we call the method on the list, the list is changed **in place**. In fact, most of these functions return `None`!

```
lst = [ 1, 2, "a" ]
lst.append("b") # adds the element to the end of the list
lst.insert(1, "foo") # inserts the 2nd parameter into the 1st index
lst.remove("a") # removes the given element from the list once
lst.pop(0) # removes the element at the given index from the list
```

Lists in Memory

Data in Memory: Mutable vs Immutable

Recall that all the data we work with in a program must eventually be stored in the computer's memory as binary.

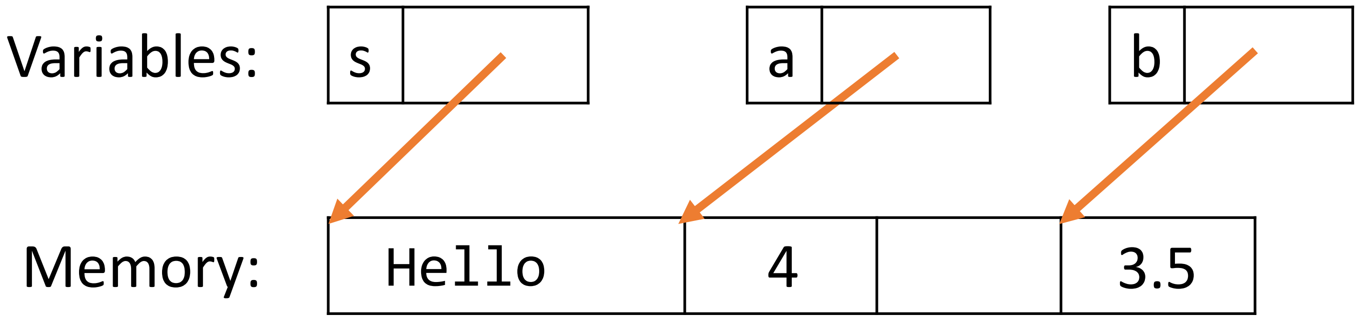
Data is stored in memory differently based on whether it is **mutable** or **immutable**.

Let's compare how memory works for strings (immutable) vs. lists (mutable).

Storing Variables in Python

When we set a variable to a value and store it in memory, the variable and the value are kept **one step apart**. The variable is set to a **reference**, which points to the place in memory where the value is stored. The values may not be in contiguous places in memory.

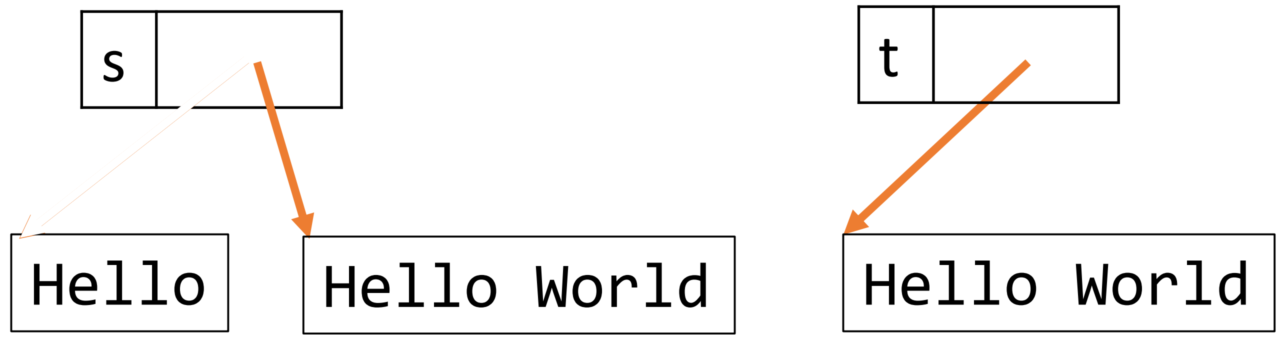
```
s = "Hello"
a = 4
b = 3.5
```



Strings, Numbers, Booleans are Immutable

When we modify the value of **immutable values**, such as strings, numbers, and Booleans, Python makes a **new value** and reassigns the variable to reference the new value.

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

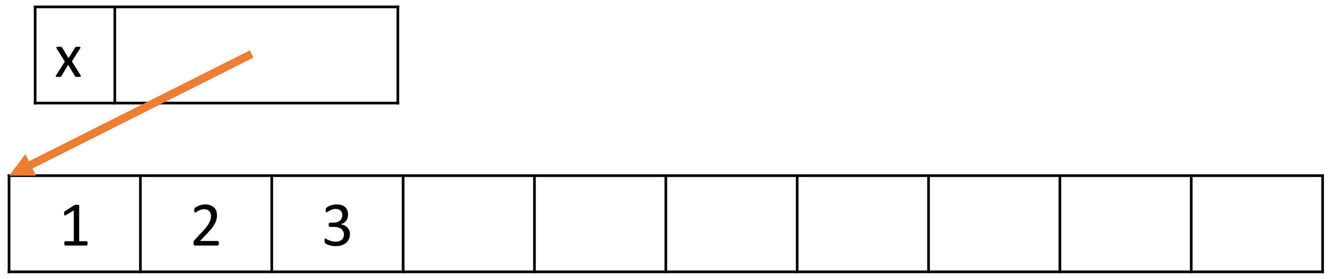


Lists are Mutable

When we set a variable to a **list**, Python sets aside a large place in memory for the list.

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]
```

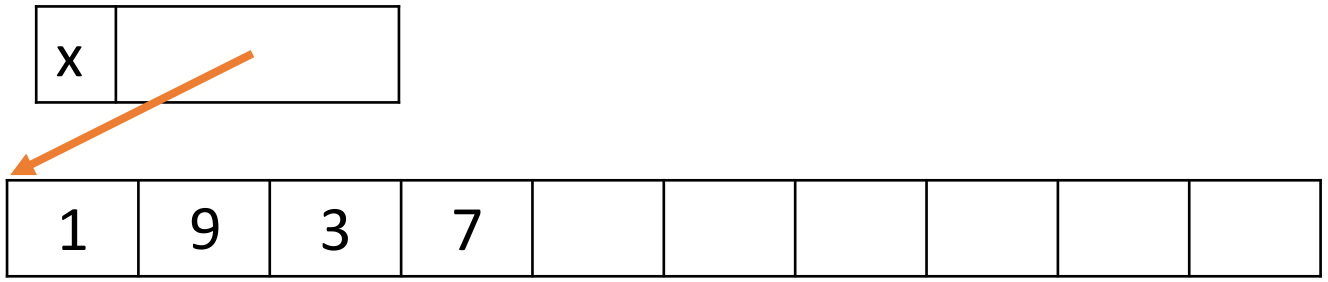


Modifying Lists in Memory

The large space set aside for the list values allows Python to add and remove values from the list without running out of room.

It also makes it easy to locate a specific value based on its index.

```
x = [1, 2, 3]
x.append(7)
x[1] = 9
```

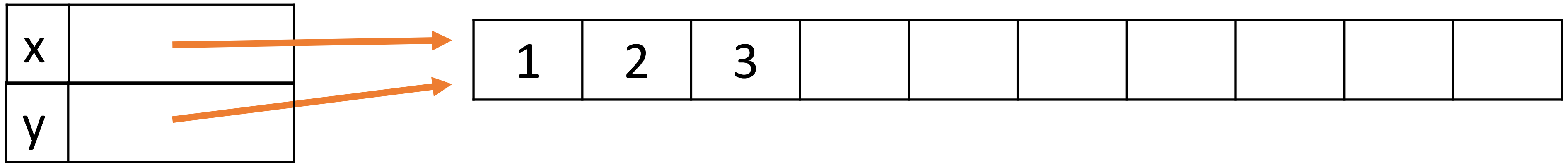


Copying Lists in Memory

The use of references causes an interesting behavior when we copy a mutable variable x to a new variable y.

The **reference** is copied, not the values. That means the **same set of values** is used for both variables. This is called **aliasing**.

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

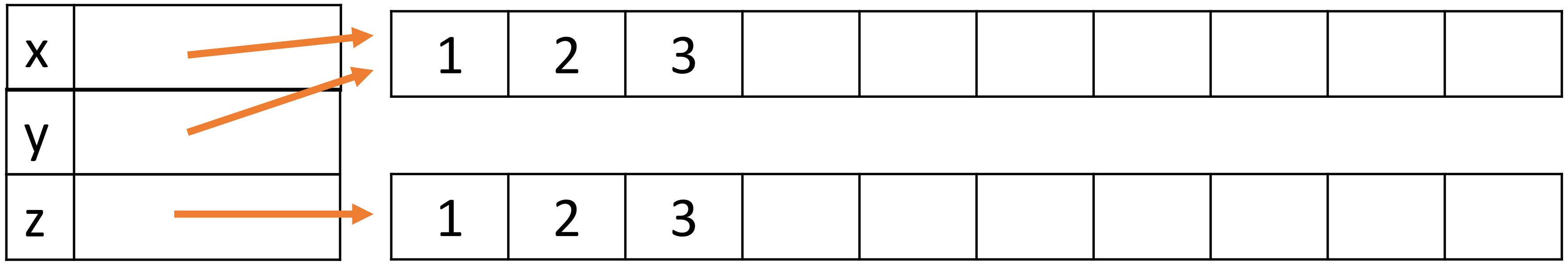


Copying References vs. Copying Values

Two variables won't be aliased just because they contain the same values. They need to **refer to the same location in memory** 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]
```

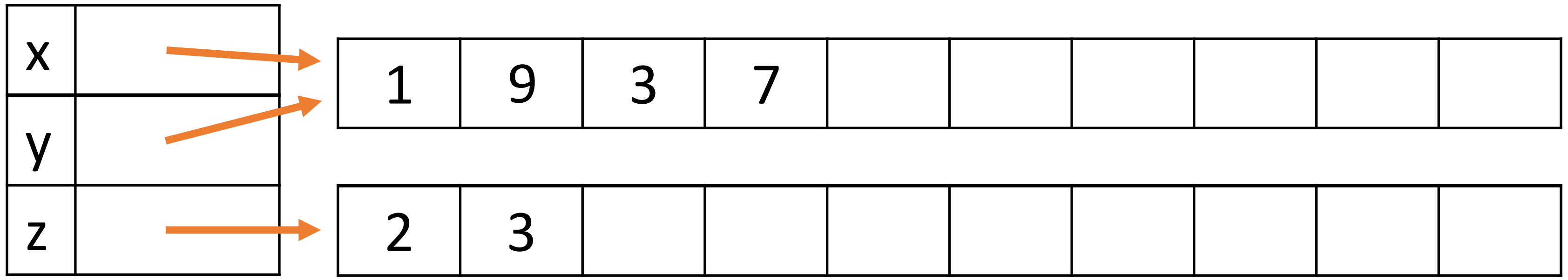


Aliased Lists Share Mutable Actions

When a mutable action is done on a list x, that action affects the **data values directly**.

Any lists aliased with x will see the same changes!

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]
x.append(7)
y[1] = 9
z.pop(0)
```



Activity: Code Tracing

After the following code is executed, what is the value of `x`?

```
x = [10, 20, 30]
y = x
x[2] = "dog"
y.append("cat")
```

Check For Aliased Lists with `id()` and `is`

If you want to check whether two variables are aliased, you can use a built-in function and an operation.

The function `id(var)` takes in a variable and returns the **memory ID** that Python associates with it. If two mutable variables have the same memory ID, they're aliased.

The `is` operation returns `True` if two variables have the same ID, and `False` otherwise.

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

print(a is b) # True
print(a is c) # False
```

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.

Two Ways to Update Values

How do we update a value in a list **destructively**? Use index assignment.

```
lst = [1, 2, 3]
lst[1] = "foo"
```

How do we update a value in a list **non-destructively**? Use variable assignment with list slicing and concatenation.

```
lst = [1, 2, 3]
lst = lst[:1] + ["foo"] + lst[2:]
```

Two Ways to Add Values

How do we add a value to a list **destructively**? Use append, insert, or +=.

```
lst = [1, 2, 3]
lst.append(5)
lst.insert(1, "foo")
lst += [10, 20] # Annoyingly different from lst = lst + [10, 20]
```

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

```
lst = [1, 2, 3]
lst = lst + [5, 10, 20]
```

Two Ways to Remove Values

How do we remove a value from a list **destructively**? Use remove or pop.

```
lst = [1, 2, 3]
lst.remove(2)  # remove the value 2
lst.pop(1)     # remove the value at index 1
```

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

```
lst = [1, 2, 3]
lst = lst[:1]
```

Activity: Which Lists are Aliased?

At the end of this set of operations, which pairs of lists will still be aliased? Fill out the Piazza poll with your answer(s).

```
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")
```

Functions Can Be Destructive/Non-Destructive

When we write new functions, we can specifically make them destructive or non-destructive too.

When a variable is passed as an argument to a function, that variable is **aliased** with the function's parameter.

That's what makes it possible for `list.append()` and the other mutable functions to work!

Destructive Functions Use Mutable Methods

When writing a destructive function, use index assignment and the mutable methods (append, insert, pop, and remove) to change the input list 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 anything, because the parameter `lst` and the argument `x` **refer to the same list**.

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

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

Non-Destructive Functions Make New Lists

When writing a non-destructive function, 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 integer list that is given. This function **does need to return** the result, as the parameter is not changed. After the call to the function, the variable x refers the original list and 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)
```

Sidebar: Don't Change List Length in For Loops

It is a **very bad idea** to destructively change a list's length while looping over it with a for-each or for-range loop.

This will often lead to unexpected and bad behavior, since 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 change the length of the list. 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
```

Why Does Aliasing Exist?

Aliasing may seem unnecessarily complicated at first. But it has a highly useful purpose: it lets us directly modify the **program state**.

This is especially useful when we think about how aliasing can be used in functions.

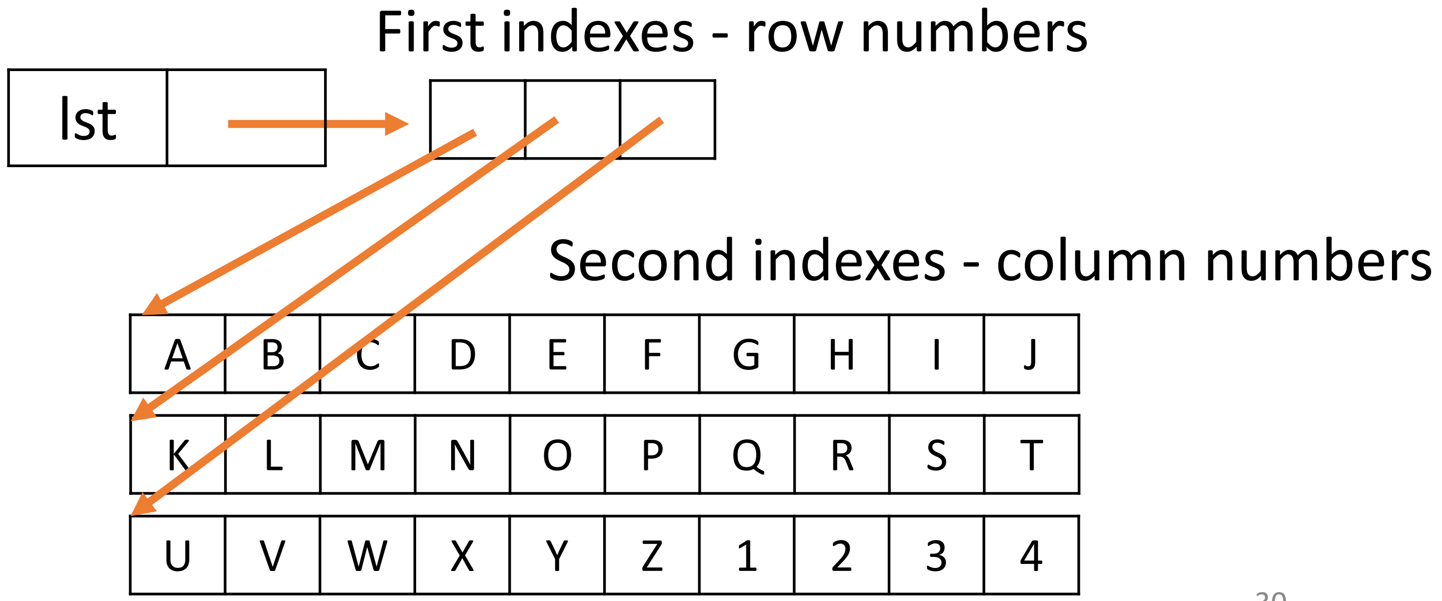
Another good reason to use aliasing is that it helps **save memory**. You don't need to copy a huge list of values over into a new place in memory every time you make a new version of the list.

2D Lists

2D Lists are Lists of Lists

We often need to work with data that is **two-dimensional**, such as the coordinates on a grid, values in a spreadsheet, or pixels on a screen.

A 2D list is just a list of lists. For example, the 2D list to the right splits the alphabet across three rows.



Syntax of 2D Lists

Setting up a 2D list is no different than setting up a 1D list; each inner list is one data value.

```
alphabet = [ ['a','b','c','d','e','f','g','h','i','j'],
             ['k','l','m','n','o','p','q','r','s','t'],
             ['u','v','w','x','y','z','1','2','3','4'] ]
```

When indexing into a 2D list, the first square brackets index into a row, and the second index into a column.

```
alphabet[2]    # ['u','v','w','x','y','z','1','2','3','4']
alphabet[1][6] # 'q'
```

Looping Over 2D Lists

When you loop over a 2D list, you need to use **nested for loops**. Often, the outer loop iterates over the indexes of the outer list (rows) and the inner loop iterates over the indexes of the inner list (columns).

```
for row in range(len(alphabet)):
    for col in range(len(alphabet[row])):
        print(alphabet[row][col])
```

Learning Goals

- Recognize how **aliasing** impacts the values held in mutable variables
- Recognize the difference between functions on mutable values that are **destructive** vs. **non-destructive**
- Use **2D lists** when reading and writing code to work on data over multiple dimensions