#7: Aliasing, 2D Lists, and Objects

SAMS SENIOR CS TRACK

Last Time

Utilize **lists** as data structures when writing programs

Use **indexing and slicing** on strings while writing functions

Understand the difference between **mutable** and **immutable** datatypes

Use **string and list methods** while writing programs

Today's Learning Goals

Understand how **aliasing** works with lists and other mutable objects

Build **lists of multiple dimensions**

Build **objects** with **properties and methods**

Aliasing

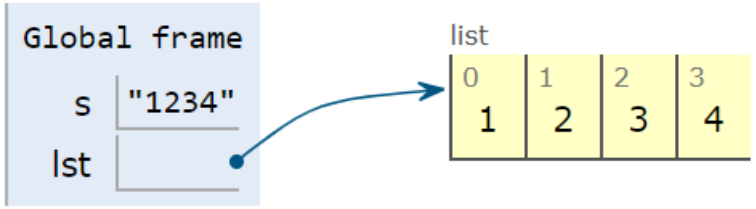
Reminder: Lists are Mutable

Last week, we learned that lists are **mutable**- the values in them can be changed directly.

This is possible because we aren't actually storing the list value directly inside its variable. Instead, the variable contains a **reference** to the list. We can change the list while leaving the reference the same.

This is why we can call `lst.append(item)` without needing to set the call equal to a value!

```
1  s = "1234"
2
3  lst = [1, 2, 3, 4]
```



Side Effects...

The mutable nature of lists has an important side effect- **copying variables works differently than with non-mutable values.**

Normally, if we make a copy of a string or number, the copy is disconnected from the original value:

```
a = "foo"

b = a

a = a + "foo"

print(a, b)
```

Side Effects...

The mutable nature of lists has an important side effect- **copying variables works differently than with non-mutable values.**

But if I copy a list, we'll get an unexpected result...

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

Aliasing

List copying is broken because our variables are **aliased**. They both store the same reference, where each reference points to the same list. Changing the list doesn't change the reference.

We can make what's going on clearer by **visualizing the code's execution**.

Example here at Python Tutor: https://goo.gl/aZwfcW

Exercise: Aliasing

Predict what the following code will print. When we run the code, check your results.

```
x = [ "a", "b", "c" ]
y = x
s = "foo"
x.append(s)
s = s + "bar"
print("x", x)
print("y", y)
print("s", s)
```

Function Call Aliasing

When we call a list in a function, the parameter is an alias of the originally provided list. This lets us write functions that are **destructive**- they change the provided value instead of returning.

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


a = [1, 2, 3]
print("before", a)
print("result", doubleValues(a))
print("after", a)
```

Destructive vs. Non-Destructive

If we want to make a list function that is **non-destructive**, we make and return a new list instead.

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

a = [1, 2, 3]
print("before", a)
print("result", doubleValues(a))
print("after", a)
```

Built-in Functions

Built-in list functions can be destructive or nondestructive.

Need to add a single element?

`lst.append(item)` is destructive; `lst = lst + [item]` is non-destructive.

Need to remove the last element?

`lst.pop()` is destructive; `lst = lst[:-1]` is non-destructive.

If you aren't sure whether a function is destructive or nondestructive, pay attention to whether it changes the list and whether it requires an assignment.

Exercise: Lists in Functions

We want to write a function `replace(lst, oldItem, newItem)` that replaces all instances of `oldItem` in the list with `newItem`.

How would we implement this function destructively?

How would we implement it non-destructively?

2D Lists

Multi-dimensional Lists

Reminder: lists can hold any type of data. That includes more lists!

We often talk about creating **two-dimensional lists**. These are just lists that contain one-dimensional (regular) lists. They're useful for storing information that comes in grids (think pixels, game boards, spreadsheets...)

```
grid = [ ["city", "state"],
         ["Pittsburgh", "PA"],
         ["Baltimore", "MD"],
         ["New Orleans", "LA"] ]
```

2D List Indexing

When indexing into a multi-dimensional list, you index from the **outside in**.

With 2D lists, we refer to **rows** (the inner lists) and **columns** (the indices within the list). This lets us treat 2D lists like data tables.

```
lst = [ [ "row 0 col 0", "row 0 col 1" ],
        [ "row 1 col 0", "row 1 col 1" ] ]
print(lst[1]) # [ "row 1 col 0", "row 1 col 1" ]
print(lst[1][0]) # "row 1 col 0"
```

2D List Iteration

Likewise, when iterating over a multi-dimensional list, we use **multiple nested loops**. For 2D lists, we first iterate over the rows (the inner lists), then the columns (the elements).

```
lst = [ [ "a", "b" ], [ "c", "d" ] ]
for row in range(len(lst)):
    for col in range(len(lst[row])):
        print(lst[row][col])
```

Exercise: 2D Lists

We want to write a function that takes a 2D list of one-character strings and a word, and returns the [row, col] index of the starting character of the word if that word can be found in an adjacent string of characters in the list, or None otherwise.

Basically, we want to write a word search solver!

Testing wordSearch

```
def testWordSearch():
    board = [ [ 'd', 'o', 'g' ],
              [ 't', 'a', 'c' ],
              [ 'o', 'a', 't' ],
              [ 'u', 'r', 'k' ] ]
    print(wordSearch(board, "dog")) # [0, 0]
    print(wordSearch(board, "cat")) # [1, 2]
    print(wordSearch(board, "tad")) # [2, 2]
    print(wordSearch(board, "cow")) # None
```

Objects

Data Structures

So far we've learned about two different types of data structures- strings and lists.

There are hundreds of other types of data structures we can design! Sets, dictionaries, arrays, trees, graphs, heaps, and more...

These all share two things in common:

- All hold a certain kind of **data** (usually in different formats)

- All can have certain **methods** performed on them

Objects and Classes

The most generic kind of data structure is called an **object**. An object has no pre-specified data or methods; you have to specify them yourself!

We can design a new type of an object by writing a **class**. A class is like a template that describes how an object should be structured. An object, or **instance**, is then a specific item of that class.

```
class Dog(object):
    # define properties and methods of a generic dog here
    pass

fido = Dog() # fido is now a specific instance of the class
```

Attributes and Methods

When defining a class, we need to determine what **attributes** and **methods** the class should have. Attributes hold the data that the object represents; methods describe what the object can do and what can be done to it.

We store attributes in **variables**, and methods in **functions**. For example, our Dog class might have the attribute breed, and the method speak().

We also need to give each class a **constructor**, which is called when an instance is first created. The constructor lets us set up the initial data in the object. In Python, the constructor must be called `__init__`.

Self

When defining the attributes and methods a class has, we need to refer to the variable **self**. This variable will be replaced with the specific instance of the class being referred to when the method is called or the attribute is accessed.

Think of it this way: a generic dog can't speak(), and doesn't have a breed. But your specific dog can speak() and has its own specific breed.

We then refer to an instance's attributes and methods with `<instance>.<attribute>` or `<instance>.<method>()`, like we do with lists and strings.

```
class Dog(object):
    def __init__(self, breed):
        self.breed = breed
    def speak(self):
        return "Bark!"

stella = Dog("husky mix")
print(stella.speak())
print("Breed:", stella.breed)
print(Dog.speak()) # will crash
```

Example: Book class

Say we want to program a catalog system for a library, and we want to design a Book class as a new data structure. What attributes and methods should a Book have?

Today's Learning Goals

Understand how **aliasing** works with lists and other mutable objects

Build **lists of multiple dimensions**

Build **objects** with **properties and methods**