CSE231 - Lab 07

More on Mutability, References, Shallow/Deep Copies

Lists are weird

If you've been working with lists for a bit, you might have noticed that they can have some pretty strange interactions when you start creating multiple variables circulated around them.

This ties into the idea of mutability, and like we've discussed, lists are *mutable*. They are one of three mutable objects in Python (the others are dictionaries and sets). Tuples are the *immutable* equivalent to lists.

Today we're going to expand upon mutability and what it truly means. Be ready, this is going to get a bit more technical.

References

When you pass around `int`, `str`, or any other immutable object in Python, you are passing along a variable's ***value*** (taking a copy). Variables have two things they keep with them at all times: an address in memory (not shown to you), and a value (the RHS of the assignment).

A variable's address is key to what the variable holds as a value, changing the address of a variable is transcendent among all changes you could make because you're literally going into your computer's memory and fundamentally changing the path your variable had to a specific value.

References are a way of storing a variable's address -- it's a path to the value. When you create a *mutable object* and start doing stuff with it, it is being "*passed by reference*", meaning that assigning a variable to another variable that holds a mutable object is actually holding the original path to the value it was given. What does this mean, then?

Mutability, Memory and References

So then, when you initialize some variable$_1$ with an ***immutable*** type, and assign another variable$_2$ = variable$_1$, your newly created variable$_2$ will *copy* the value of variable$_1$, meaning that changes to variable$_1$ will not affect variable$_2$ and vice versa.

If variable$_1$ was instead a ***mutable*** object, changes to variable$_1$ ***will also change variable$_2$*** and vice versa. Assigning mutable types normally like this is called ***shallow copying.*** You're not passing the values along to another variable, you're passing ***the paths to the values (references)***.

Deep Copies

So what if you don't want to change your original list? Well, you would need to take what's called a ***deep copy***, which is what traditional assignment does for immutable objects. You have to deep copy *mutable* objects because ***this will copy the values instead of the references.***

To take a deep copy, you have to `import copy`, and use the function `copy.deepcopy()`. Deep copies iterate through every element in your mutable object, and take copies of all of them to bring you a new, unique mutable object. The are no more references.

Mutability & Functions

Because mutable types pass by reference, this, for better or worse, bleeds into scope logic. Passing a mutable object into a function as a parameter, again, passes a ***reference*** to that mutable object unless you specifically take a copy beforehand. Changes to the reference of the object inside the function will change the object in ***every scope***, including the global namespace.

One of the reasons that tuples exist is so that you don't accidentally modify the lists you want to stay constant. If you were to try and modify a *tuple* in your function, it obviously wouldn't work.

Lab time, this one sucks btw lmao