[SQUEAKING]

[RUSTLING]

[CLICKING]

ANA BELL: So let's get started with lists and mutability. So last lecture, we talked a lot about what it means to have these mutable data structures, lists. Today, we're not off the hook.

We will continue talking about the idea of mutability, but we're going to do it in the context of removing items from lists and some of the pitfalls that come with that. And then we'll go into, along the way, ideas about cloning or making copies of lists, and aliasing, making another name for the same object in memory.

So first, let's quickly talk about making a copy of a list because so far, when we were dealing with these mutable objects, we notice that it's sometimes inconvenient to have to mutate the list, right?

And it's hard to keep track of the fact that we're mutating a list. And there are some problems for when it does make sense to make a copy of our list so that we can mutate the copy or mutate the original while still having those original items saved somewhere else. OK?

So you can ask Python to make a copy of a list. And basically, behind the scenes, it creates a new list object for us in memory and copies over every single element from the list you'd like to copy into the new list. OK?

So the syntax for doing a copy of a list is as follows. So we've got a list that's already made called L. And we want to make a copy of it. So the syntax is L squared brackets with a colon inside it. And behind the scenes, Python makes this list inside memory.

And then we save that new list that has the exact same elements as L into a list named Lcopy. OK. And so in memory, the way this looks-- so if I have this code here where I name my list Loriginal-- again, I'm choosing a different name than L just to show you that whatever list object I have, that's the name I need to reference.

So if I have Loriginal is 4, 5, 6 in memory, if I want to make a copy of my list, I just say Loriginal square brackets with a colon inside it. That means copy every single element from beginning to end of this list and bind it to the name Lnew.

So notice-- in memory now, I have two list objects. They're referenced by different names. And so if I change one of them, the other one will not change, right? They're now completely separate objects.

OK. So we're starting this lecture off with a quick little exercise just to get you to remember what we did last time and to practice writing a little bit of code with mutable objects. So I would like you to write this function called remove_all.

This is going to feel very similar to something we did last lecture. So last lecture, I asked you to write a similar function, which took in a list L and an element E. And that function from last lecture created a new list and then basically populated that new list with all the elements-- it had all the same elements as L except for omitting the ones that were equal to E.

OK. This version that I would like you to write for me is not going to create a new list and return this new list. It will mutate my input L such that you're going to only keep the elements from L that do not match E.

OK. So I'm going to give you a hint for how to do this. So the process for this is going to make use of this thing that we just saw, which is, I want you to first save the list as is into a copy. And then at the end of last lecture, we saw a way for us to mutate a list, to empty it out of all the elements.

So we still have that object in memory, but we're just essentially clearing it out. We remove all the elements from it. So first, make a copy and save the elements. Then clear the list we want to mutate-- L.

And then iterate through the copy and add all of the elements that do not equal E back into L. So that should be the process. And in the end, when we call this remove_all function, the thing that we're passing in will have been mutated. We don't have anything to return.

We're just mutating the thing that's being passed in. So I'll give you a couple minutes to work on that. And you can start writing it around line 30.

OK. Does anyone have some code to start with? Yes.

STUDENT: Initialize new list?

ANA BELL: Yup. Initialize a new list. Yep. What do you want to call it? Lnew? Good name. Lnew equals-- how do we make a copy, and what do we copy?

STUDENT: Why do we copy [INAUDIBLE] added to the ones that aren't equal?

ANA BELL: So what we'd like to do is mutate L, right? But L already contains a bunch of items in it, right? So that's why we first want to make a copy of it, right?

So just like in the syntax from the slides, this will essentially save for us everything that we already have in L in a new list called Lnew. OK. So now that we have that, does anyone-- yeah.

STUDENT: I used L.clear.

ANA BELL: So L.clear does not take a parameter in, right? It's just a function that empties out L fully. So it'll basically drop every single element in L. OK? But we will see a function that will remove elements.

OK? So if we do L.clear, then L becomes the empty list, right? L just becomes this. So now that I've mutated my object to contain none of my elements in it, how do I add back in the elements that satisfy the condition?

Yeah. So for n in Lnew-- so I'm iterating over the list that actually contains stuff, the thing I've copied. And then--

STUDENT: You can just be like, if e not equal to n, then you want append [INAUDIBLE].

ANA BELL: Yeah, exactly. L.append. So notice-- I am appending to L, but I'm iterating over Lnew, right? Lnew has all of these elements in it. I want to touch each element to see what value it has.

If it's not equal to the one from the parameter e, then I add it to my list, L, the one that's currently empty. OK? And then do I need to return anything? We don't need to return-- it won't hurt to return L, but L will already be mutated by virtue of this function.

So we don't need to return any L. Right? L is my parameter that I've passed in. So there's nothing to return. It's just being mutated in the function. So when I make my function call here, I'm passing an L in. I'm just making a call to remove all with this Lin object, which is this one here.

And notice-- I'm not saving the return from this function to anything, right? Because this function will just mutate whatever I passed in. And then if I just print the value of Lin after this function call, it'll print the mutated value.

STUDENT: So when we're appending n [INAUDIBLE]?

ANA BELL: Yes, sorry. We should append. Thank you. Yep. And that looked weird. Perfect. And so if I run the other two examples here, I'm removing 1. So it should just show me a list with all 2. And here I'm removing 0. And 0 doesn't even exist, so it doesn't mutate that input list at all.

OK. So now, we can start talking about other operations on lists which deal with removing lists and making the lists smaller. So we're actually going to take elements away from the list. And this is similar to what the suggestion was instead of to clear out a specific element, right?

But the clear function removes all the elements. However, these functions will remove certain elements from our lists. So there's three different ways that are on this slide. And I'm going to show you an example with this list L, showcasing what each one of these functions do.

But first, I'll just explain them. So one option for removing an item from a list is if you know the index of the item you want to remove, like you want to remove the very first one in the list, or the last one in the list, or the halfway point, or something like that, you can tell Python to remove the item from list L at a particular index with this del parentheses-- so this function del. And you pass in L at whatever index you want to remove.

Now, sometimes, you want to remove the item all the way at the end of the list-- so the farthest-most right. In that case, there's an operation called pop. And you call pop on list L. So if you just say L.pop with nothing in the parentheses, Python will automatically grab that last value from the list and drop it from the list.

OK. Now, pop is a little bit interesting because it has a return value. We're using this dot notation, which we used with append and clear and a bunch of other things from last lecture. But here this pop-- not only does it have the side effect of mutating my list by dropping the last element from it, but it also returns something.

So this function call here will return for me the value of the element that got dropped, just in case you want to do something with it.

And lastly, if you know element you'd like to remove specifically, so if you have a list of a bunch of names and you want to remove Anna from that list and the string Anna is what you'd like to remove, you do that using the function L.remove.

So whatever lists your names are part of, you say that list.remove. And then you pass it in the string Anna or the number 5 or whatever actual element you'd like to remove. OK.

Now, if there are many elements that match that value, if there's many Annas in my list of names, it will only remove the first one it finds-- so from index 0. All the other ones will remain. You'll have to call that function again.

So let's look at this example here. I've got this list of seven elements within it. Let's do a few of these operations all in a row. So each one of these operations will mutate my list. So the operation right after it will work on the mutated list.

OK. So let's start with this L. If we say L.remove 2, Python will look for the element whose value is 2. Well, there it is. It's at the front of my list. That's fine. And Python will remove that element. So this list will now be one element less shorter, right?

And that 2 is going to be gone. So the list L will now be mutated to be 1, 3, 6, 3, 7, 0. All right. Well, what if we remove 3 now, right? So we've done the operation to remove 2. We've ended up with this mutated list. Now, what if we remove 3 from this mutated list?

There's two of them in there, right? The element that's going to be removed is the first one it finds-- so just this one here. And again, this is an operation that mutates my list. So this list here that I've started with would be one element shorter. And that 3 will have been removed, right? So now, I've got 1, 6, 3, 7, 0.

All right. What if we want to delete an element at a particular index? So now again, we're working with the mutated list-- 1, 6, 3, 7, 0. This del function takes an index in a specific list and removes the element that is there. So in this case, I want to remove the element at index 1.

So in this list here, the element at index 1 is the 6, right? This is 0. This is 1. So the 6 will be removed. And my list will be mutated to just contain these four elements-- 1, 3, 7, and 0.

And lastly, if we pop, that function will just remove the element at the end of the list. The element at the end of the list is this 0. So the list through the side effect of pop is going to be mutated to contain just the three elements except for the last one. So it'll contain 1, 3, and 7.

And additionally, if I'd like to save the value of the element that got removed from the end of the list, the 0, you can because this function call here-- L.pop-- you can save the return value into a variable.

None of the other ones-- del or remove-- have any return, right? So if you saved a variable from the function call to a variable, that variable will be none. Pop is special because it actually grabs that variable value and returns it. OK.

So all of these operations mutate the list, right? So that means as we did operation after operation, we were working with the mutated list. OK. Yes, there was a question.

STUDENT: Yeah. It should still be L1-- the third one to the left.

ANA BELL: I'm sorry. Say it again.

STUDENT: In the third one, you said delete L1.

ANA BELL: L at index 1. Yeah. So the L at index 1 here works on the list we have just mutated. So this one. The element at index 1 is the 6.

STUDENT: Sorry.

ANA BELL: Oh, yeah. Yeah. No worries. OK. So let's look at the code we just wrote in the you try it exercise. And try to rewrite it using this remove operation. Well, the way we can think of it is we'd like to remove the element that is e, right?

So we know the value of the element we'd like to remove. It's 3, or 5, or 1, or 2, or whatever. So that's e. And we know of an operation that can remove the element from the list. It's called remove, unsurprisingly.

So what we can do is we can say, L.remove e. right? And that would remove the first instance of the element in the list, but I might have many of these elements in my list. So we can just write a little while loop around this operation.

And we say, while we still have this value in our list, remove it. Right? So that's what this while loop is doing. e in L is going to be there true or false, whether the number 5 or whatever is in my list. And as long as I still have a 5 in my list, call L.remove on 5 or whatever it is. So nice little two-liner here to solve the same problem.

Now, what if we rewrote that code in a slightly different way, again using remove, but let's say maybe we didn't realize we could use a while loop? And instead, we used a for loop to iterate over each element in L. And if that element is equal to e, remove it. Right? Seems reasonable.

So what would happen? And I can run it for you guys. So if we run it with this code here, this is the one from the slides, just to show you that I'm not making it up. So if this is the code that we wrote, I tried to remove the 2 from the list.

And when I printed the result, it actually printed 1, 2. So I have two elements left in my list. It looks like it didn't correctly remove a 2. And at first, it's surprising why this is, right? Because the code looks right. It seems to work just fine.

But let's step through this memory diagram and see exactly what happens step-by-step, so with each iteration of our for loop. So originally, I've got L containing 1, 2, 2, right? So far, so good. That's just us doing this line here.

And then I make a function call to remove all. So I want to remove the number 2 from my list. I've got a for loop, where my loop variable is called elem. And it will iterate through each element in my sequence where my sequence is all the elements in L.

So first, it will be 1, right? Then it will be the next value in the sequence-- 2, and then 2, and then 2. Right? So here I've just got elem initialized to the first value in the sequence. If elem equal equal e, well, the 1 does not equal the 2. So then we do not remove anything.

Next, the for loop goes on to the next value in my sequence-- the 2. So now, elem is 2. And if elem equal equal 2, it does equal 2, what am I going to do? Well, I need to do L.remove e. So this is where bad things happen.

I'm going to remove an element from my list, right? So I still have those three 2s in there. But as soon as I drop one of the 2s, all the elements beyond that 2 shift over. But Python doesn't know that it should also shift over the pointer. Right?

It's still pointing to that element that it's currently at. It's not going to shift itself backward just because you removed an element. And so Python just finished removing the element. And now, it says I finished this loop through.

So I need to go back up here and make element be the next value in my sequence-- the next 2. So I've essentially skipped over one thing that I needed to remove because when I remove the item, everything else shifted over as well, but my pointer didn't decrement.

OK. So this is a big problem. We can finish off here, but we've already seen the problem. The last time through the loop, Python sees-- well, is this 2 equal to the thing I want to remove? It is. So it removes it.

And this is the end. It has no more values left to go through in the sequence because its pointer is already out of the bounds. OK. Is everyone OK with that issue? OK.

So the problem here with remove is that we're iterating over a list as we're mutating it. Right? And so removing these items can cause unpredictable behavior. Something like this could still happen if we were adding items, except that we're usually adding items to the end of the list with append.

If we were adding items somewhere in the middle or somewhere around where our pointer is supposed to be, I think we could theoretically run into the same issue when we're adding items where we might skip elements or we might see an element twice. OK? It's just more apparent when we're removing items.

So this is the big thing that we're going to talk about in this lecture. So I'm going to go through another example. This is tricky example number 4, where we're going to do a very similar thing, but we're going to have a loop iterating over L's elements directly, just like we did, but doing a different task just so we're not doing that same remove_all task.

So let's look at a slightly different problem. This will be in the context of a function called remove_duplicates. This function will take in two lists. So as an example here, I've got a list with 10, 20, 30, and 40 in it. And I've got another list with 10, 20, 50, and 60 in it.

The purpose of this function is to mutate L1, OK? And the way I want to mutate L1 is such that if an element in L1 is also in L2, I want to remove it. All right.

So the 10 and the 20, notice, are common to L1 and L2. So I would like to remove the 10 and the 20 from L1. The 30 and the 40 stay because there's no 30 or 40 in L2.

So that's our task. And this is the code that supposedly does this. So I've got a loop that goes through each element in L1-- so 10, then 20, then 30, then 40. And I ask if that element is in L2. So here they are. There's two of them here. Then remove it from L1.

Very similar thing to what we just did. This code doesn't work because if we actually run it, in the end, Python will mutate L1 to contain the 20, and the 30, and the 40, right? Whereas we only want it to keep the 30 and the 40 because the 20 also appeared in L2. So why in the world did we keep it?

Well, we kept it because of the same issue that we just saw, where mutating a list as we're iterating over it and we're doing a removal-- so we're again skipping over an element. So let's just step through this one just to show you again what can happen.

So here I've got 10, 20, 30, 40 for L1, and 10, 20, 50, 60 for L2 in my loop. My variable is e, so first, it'll be 10. And we ask, if 10 is in L2-- that's true-- remove it from L1. So you can see what's going to happen. My 10 is removed. Everything else shifts over by 1, but my loop index stays fixed.

Next, Python says, I'm going to increment my variable e to go to the next item in my sequence. So e becomes the 30. And already, I've skipped over one element that I was interested in removing.

So here when we're pointing to the 30, Python says, well, the 30 is not in L2. So we don't do anything. And then it points to the 40. The 40 is not in L2. So we don't do anything. And then the code is done. And we've erroneously finished with mutating L1 to just be the 20, 30, and the 40.

OK. So let's try to rewrite the code to actually work by using copies. So we certainly could use the same trick we did with the first you try it exercise, where we could make a copy, clear L1, and then add the elements back. We could do that.

But we can also do a slightly different version of that where, again, we make a copy. So here I've got L1_copy equals L1 square bracket colon. And then the key thing here is we're iterating over the copy, right?

So if we iterate over the copy, we're not going to mutate the copy, but we will mutate L1. So for the for loop, variable goes over the copy, but the removal is done from L1. So to visualize that, this is what happens. So I've got L1 and L2 as before.

So when I make my function call here, I have L1_copy equals L1 square bracket colon. So this makes for me a new variable inside memory, which is an exact duplicate, copy or clone, of L1. OK.

So every one of my elements is now saved. So I can do whatever I'd like to L1 and know that I can still have a way to iterate and look at each variable from the original L1. So now, my loop variable e goes over elements in L1 copy.

So first, we look at the 10. And I say, if the 10 is in L2-- it is-- remove it from L1. So notice-- I have just mutated L1, not the copy, to be one element less.

Then the loop variable e goes to the next value in my sequence. So I'm not skipping anything here because I didn't mutate L1_copy. So now we look at the 20 correctly this time, right? So now we ask, is the 20 in L2? It is, so we remove it from L1.

And then the 30 and the 40, we do nothing. Questions about this? Is this OK? Is this too fast? Is this too slow? Good. OK. OK.

So that's using copies, a.k.a. clones to help you keep track of values in an original list without overwriting them or without removing them accidentally. Now, I want to talk about aliases because this is a very important topic when we have these mutable data structures.

So let's do a quick overview of what an alias is. So if we think about a city-- for example, Boston-- an alias for Boston is basically any other name that refers to the same city, right? The same object.

So Boston, also known as the Hub, or Beantown, or Athens of America-- all of these names refer to the same inherent city, right? So if I say Boston is small and tech-savvy, then those two attributes or properties refer to the object itself, right? The city.

So the Hub is small and tech-savvy, or Beantown is small and tech-savvy, right? It doesn't matter what name I refer to this object as.

The same set of properties still apply to it. And so if I add an attribute or if I take away an attribute through one of these aliases, through one of these names, well, if it's suddenly snowing in Boston, then yes, it's snowing in the Hub or it's snowing in Beantown, right? Because these are just names for the same object. And so that idea is also something that comes up when we deal with these mutable objects.

If you don't explicitly tell Python you'd like to make a copy of a list and you just use the equal sign between a mutable object and another name for this mutable object, then Python only creates an alias for that object.

So notice-- we had to say explicitly, I want to make a copy with the square brackets colon. If we write code that looks like this-- so here the only difference I've done-- so the code on the right is the one that worked. The code on the left is me not making a copy of my L1. I'm only using the equal sign directly.

And in Python, using this assignment operator, the equals sign, means that you're making an alias for that same object in memory. So it's just another name to refer to that same object. If you mutate that object through L1, L1_copy will also have been mutated because it's pointing to the same object and vice versa.

So really, this particular code on the left here is not any better than saying for e in L1 because L1_copy is pointing to the exact same object in memory. OK. So let me show you exactly what this means in the little cloud diagram that we've been doing.

So this is the code that creates an alias, not a copy. So I've got L1 equals 10, 20, 30, 40. L2 is 10, 20, 50, 60, just like before. The code up here-- so L1_copy equals L1-- I just named it copy, but it's not actually making a copy, right?

Because nowhere did I say explicitly to make a copy using the square brackets colon. So the alias in memory means that it's just another name pointing to that exact same object. OK.

So then the for loop for e in L1_copy is iterating through this object here, which is being pointed to by L1_copy and L1. OK. So if I'm iterating through and removing elements as I'm doing so, this is just the original buggy code that we had that iterated through L1, right?

So I'm removing the 10, incrementing the e variable to the next element, and then not doing anything with the 3 and not doing anything with the 40. Does that make sense? Aliases? Is that all right? OK.

So the big idea that will tie a couple things together is related to functions-- formal parameters and actual parameters. So when we make a function definition, the parameters inside the function definition are called formal parameters, right?

We're just writing the function assuming that these will eventually get some actual values associated with them. When we make a function call, that's when we pass actual values. And when we have mutable objects being passed into a function, the formal parameter actually becomes an alias for the actual parameter in the function call.

So here's our function once again. The difference between what we've been seeing so far-- this is the code that we had just seen. The difference that I've done in this particular code is not named this L1 and L2 like it was named up here, right? Because it doesn't have to be named L1 and L2.

I named it La and Lb. And this will bring the point home. So when I make my function call to remove duplicates with La and Lb, Python takes this object and this object and passes them in as parameters.

So in my memory diagram, I've got La is 10, 20, 30, 40, and Lb, 10, 20, 50, 60. Right? That's what I have down here. As soon as I make my function call, remember-- Python maps out formal parameters to actual parameters.

But when we're dealing with these mutable objects, L1 and L2 are aliases for the things being passed in. So L1 will point to-- you tell me.

STUDENT: La?

ANA BELL: Yes, exactly. So here I've got the same name for the same object. And L2 will point to Lb, right? Two different names pointing to the same object. And that's why when I'm iterating through and doing whatever I am doing to these formal parameters here, Python actually mutates the objects that were passed in. Yes.

STUDENT: So does that mean that La and L1 will have the same ID?

ANA BELL: La and L1 will have the same IDs. Yeah. Yeah. Exactly. Yeah. Using that ID function we did last time. Exactly. I invite you to try it, too. But I think they should because they're modifying the same object.

Everyone OK so far? Two names, aliases for that same object. And so that's why when we're mutating L1 here, this La and Lb that we passed in will be mutated, right? So here's my L1_copy as well. So I've got three names for this particular object.

And then we do the thing where we mutate the thing, right? And then at the end of the function when it's done, this entire thing has no return. It returns none. But when we print La, the thing we're printing is this object here. It's whatever La points to. And it's this thing that was mutated through L1.

Yes, no? Thumbs up, thumbs down? Is it good? This is very cool, you guys. OK. This was a nice loose end to tie up. OK. So the last 10 minutes, I want to talk about what happens when we have lists that contain lists themselves, right?

So so far, the examples we've been working with are lists that just contain strings, or integers, or things that are immutable. But what exactly happens behind the scenes when we have elements that are mutable themselves?

So we're going to do an example. We're going to go through all of these slides working through an example where we start out with this old list that looks something like this. So we have a list that contains three elements, right? The first one is another list. The second is another list. And the third is another list.

I don't care what elements those lists have for now. All I know is at the top level, old_list contains three things. OK. So let's do aliasing. And then we'll do a shallow copy of this list. And then we'll do a deep copy of this list and show you what happens.

So in this example, what we're going to do is just a straight-up alias of old_list. So we're going to make old_list and new_list be aliases for the exact same object-- this thing here. So I do that with just the plain old assignment operator.

So inside memory, the way we're going to represent this old list is here is my list with three elements in it. And because each element is itself a list, so a mutable object, I'm not going to plop it in here, but instead, Python generally tends to make a pointer to that mutable object somewhere else in memory.

You'll see why in a couple of slides. But for now, it will look cluttered if I did so. But for now, it helps to visualize the structure. So old_list contains three elements. And each one of those elements are pointed to over here.

So if I say new_list equals old_list, Python will make another name for the same thing in memory. When I do this line here where I index new_list at index 2-- so that's 0, 1, 2-- and then I follow it to index 1 over here, so this guy here, the 6-- I have changed the string foo to be 6.

And now, new_list and old_list both are pointing to the same object. So it will have been mutated to contain that 6 in that sublist. OK. So that's aliasing.

Now, what we can do is we can create copies of mutable objects. And we can create either something called a shallow copy or a deep copy. The shallow copy is equivalent to what we've been doing with the square brackets colon. And that's perfectly OK if we're dealing with lists that just contain immutable things.

But as soon as we create a shallow copy of a list that can contain other lists or other mutable things, interesting things happen. Only the top level gets copied, but anything that's mutable at a deeper level than that top level does not get copied because if it did and you had many, many levels deep of all these mutable things, that would be a lot of things for Python to copy.

OK. So what we're doing with this particular code is we're going to create this old list here. So it's 1, 2 as the first element, 3, 4 as the second element, and 5,6 as the last element.

We're going to create something called a shallow copy. And we could have also said old_list square brackets colon. It would be equivalent. And in this shallow copy, Python only creates a copy of the top level.

So notice-- new_list is pointing to a list with three elements in it, but anything that's at a deeper level than that top level does not get copied. So all these mutable things that are my elements-- this list, and this list, and this list-- these are three mutable elements.

They do not get their own copies because we've only made a shallow copy. So what this means is that the top level-- sorry. So this is just what it prints out. So at the top level, we can add elements to old_list. And it won't interfere with the top level of new_list.

So as an example here we're going to add this 7, 8 list to old_list. So we follow old_list. And we add another element to the end of it. OK. So there it is. But that element didn't get added to new_list. Right? Because we only added it to the top level of old_list.

So now, question is, what happens if we go in and mutate one of these three shared items? So old_list and new_list is as we would expect. So let's do one more operation. So in addition to appending the 7 and the 8 like we do over here, let's also mutate one of those shared items.

So here it is. This is what we just did on the previous slide. There's my 7 and 8. And now, let's go into old_list at index 1. So 0, 1-- that's this middle one here. And at index 1 in that-- so that's 0, 1, 4 over here. Let's change the 4 to the 9.

OK? Well, when we print new_list, we're going to be printing a list with three things in it. The first one is the list 1, 2. The second one is 3, 9. We just mutated that. And the last one is 5, 6.

And when we print old_list, this one will also have that 9 over there because those middle elements are shared, but we will also have an extra element at the top level-- the 7, 8 that we just added only to old_list. OK. Thoughts on this example? What is confusing? Yeah.

STUDENT: Could you explain why the 9 gets added to the new list?

ANA BELL: Yeah. Why does the 9 get added or get changed to the new list? Yeah. So the operation called copy from this library, which is also named copy, only creates a shallow copy of the list.

So a shallow copy means that if you have a list with some elements within it-- so here in this case, we have those three elements in it-- all you're doing is copying the top structure, right? So this structure here.

But if you have any elements that are themselves mutable, they don't get their own copies. So really, inside the memory, if this one is pointing to some object like it does to that list 1, 2, the copy is also going to point to that same subobject, substructure.

And so if you're mutating this substructure through one name, if you're accessing it through the other name, that other name is still accessing the thing that was mutated. Does that make sense? Is that OK? Yeah.

And so this shallow copy is just copying the top structure here. So you can see at the top level, we have these two different lists. So that means to this one, I can add another item to the end of it, right? And that item will not be duplicated up here because this is one thing. This is another thing.

But the middle ones or any levels that are beyond that top level are shared. They're not copies. Yeah.

STUDENT: So if you edit it through the new list, is it mutual?

ANA BELL: Yes. Yes. Exactly, great question. So if we edited this number 1 here through the new list, then yeah. The old list will still see the edits because they're both pointing to these shared things. But if I edit the 7 and 8, it will only be edited through old_list because that's 7, 8 is only seen by old_list.

OK. That's basically what I've said here. And so if you really, really, really want to copy every single mutable object or every single object at all the different levels, we would have to create something called a deep copy. So we do this using copy.deepcopy. OK?

And so this is the exact same example, except that we've just changed copy.copy to copy.deepcopy. And so here we've got our old list exactly as we had before. And if we deep copy old_list, Python will make copies of every single object at every single level from old_list. So everything becomes its own object.

So now, if I mutate old_list to append 7 and 8, that only gets added to old_list. And if I mutate old_list to have this element be a 9, that only gets mutated through old_list. So old_list contains the changed values, but new_list remains untouched because I've made copies at every level. Yes.

STUDENT: So is copy.deepcopy doing old_list square bracket with the colon in the end?

ANA BELL: Yes, but then it goes further down at every single level. So the regular copy.copy does the square bracket colon. And the deepcopy goes further to all the other levels.

OK. So lots of ideas in this lecture. And last, I would highly suggest going through the Python tutor and all these examples just so you see them in a different way to see exactly how it'll be the same memory diagram that we've done, except through the Python tutor.

So it will be very helpful for you, I think. I would give that a try as you're studying for the quiz. I think what's important to realize is that we have objects in memory. And we have names that point to these objects.

And so if you get that and keep that straight in your mind, it will be very, very helpful to understanding what's an alias, what's a clone, when you're iterating over certain objects, and things like that. And the big idea here is just side effects, OK?

Every one of these operations has some sort of side effect. And it's important to make sure that you're not changing something you don't want to be changing. OK.

I guess I just had one last thing to say about lists and tuples. We've seen both of them. When do you want to use tuples and not lists? When you want something that shouldn't be changed.

So if you have something that might accidentally get changed, do not save it as a list. OK. And then on the other side, why would you use a list but not a tuple? You would use a list because you don't want to be creating copies all the time.

So when you have, again, these large databases, every time you want to make a change to it, you don't want to make a copy of everything with that small change in it. And so mutating an object is good from that respect.

OK. So that wraps up lists and mutability. Next lecture, we'll just tie up a bunch more loose ends. And then we'll get into a new topic.