[SQUEAKING] [RUSTLING] [CLICKING]

ANA BELL: All right. So hello, everyone. Let's get started. This is lecture 10.

So last lecture, we introduced two new data types. We talked about a data type called a tuple and a data type called a list. So today, we're not going to talk about tuples anymore because they were pretty straightforward.

A lot of operations you could do with strings, you could do with tuples. They were immutable objects. That means once you created them in memory, you couldn't do anything to change them. And so they were, I guess, pretty boring except that you could populate tuples with objects that were of any type. So you could populate a tuple with integers, and floats, and booleans, and other tuples, all at the same time.

We introduced lists last time as well as something that was really similar to tuples and strings in terms of manipulations. Lists were also nice because you could populate them, again, with any kind of data objects, just like you could tuples.

Today what we're going to focus on, though, is the idea of mutability when talking about lists, which is something new. We have never talked about this idea before. And so this lecture is going to be pretty heavy on that idea, and a little bit heavy on syntax and things like that to remind you of how to manipulate these compound data types. So please, if there's any questions, feel free to stop me and then I can go over what I just talked about, if there was anything confusing.

So this slide is basically a copy of the slide we had on lists last lecture. It shows a bunch of different-- oops. A bunch of different operations that you can do with lists. They're very similar to operations that you can do with strings.

So here, I'm just creating an empty list. I'm creating a list with a bunch of elements in it. So here, we can see that this list contains four elements and they are all of different types. This is an integer, this is a string, this is an integer, and this is another list. And that's totally OK to do with these data types.

Doing all of these operations, getting the length, indexing, slicing, concatenation, getting the max, all that should be review, as well as iterating a for loop over the elements in a list directly. So just like we iterated a for loop over characters in a string, this loop basically makes e, our loop variable, take on the value of every single element in our list L.

What's new? The bolded thing here is something we haven't been able to do before. And this basically goes into memory and changes the element at index 3 and L. So that's kind of how we read that. And it changes that element to have the value on the right-hand side.

So this is kind of-- I mean, we read it in the same way as we do other assignment statements. We look at the right-hand side and we evaluate that. In this case, it's only a 10.

But the left-hand side looks different. It's not a variable name as we have seen before. But in fact, it's referencing the item in list named L at index 3. So that would be index 0, 1, 2, 3. This line of code down there, L squared brackets 3 equals 10, basically replaces this entire element here with the number 10.

So on the next few slides, we're going to talk about what exactly this means inside memory, because it's different than what we've been able to-- what we've been doing before. So what exactly happens when we go into a mutable object like a list and we change an element using this exact syntax?

Well, let's draw our memory diagrams the way we have been in the past. Here's our little cloud representing the memory. L equals 2, 4, 3, creates this list for me in memory, this list object. And L is the name that I'm referencing to this list object right. So I'm basically binding the name L to that object in memory.

L squared brackets 1 equals 5 tells Python to follow the name L to the object in memory and then look up the index in the square brackets. In this case, index 1. So that's 0, 1, this 4. And take the element at this location and override it to be whatever the right-hand side says. So the right-hand side says five, so basically we're going into memory and changing that middle element.

So this is different than strings and tuples. We were not allowed to do anything like this with strings and tuples. So let's look at an example on the next slide about what this means.

But the idea here I'm trying to get at is this object that we have changed one of the elements inside-- for which we've changed one of the elements, we've changed the object itself. We didn't make a new copy. We didn't make a version of that object. We have changed the object itself.

So let's see maybe similar code that might have-- that you might think does the same thing, except with tuples. So the first two lines of code are going to be the same. We've got L in memory being the object 243, or L being the name bound to the object 2, 4, 3, L, square brackets, 1. So L at index 1 equals 5 changes that middle element to be a 5. Same as the previous slide.

Now what if we had these two lines of code? t is going to be a variable name that's bound to the tuple 2, 4 3. So notice this is now the tuple denoted in parentheses. If I say t is equal to 2, 5, 3 what happens?

Basically, with this line, I am creating a new object in memory. So there's my new object. And I'm taking the name t and I'm binding it to this new object.

The old object, 2, 4, 3 as a tuple, still remains in memory. I have not modified that object at all. It's still there, I've just lost the binding to it.

So the name t is separate from the actual object in memory. In terms of tuples, what that means for us is we can never change the tuple object in memory once we've created it. But with lists, using this specific operation, this one right here L square brackets 1 equals 5, this does allow us to go into memory and literally change that object that is associated with the name L.

Is everyone OK with this slide? Does this make sense? This showcases the difference. So you need to think about what is the name of the object versus the object itself in memory.

So that shows you how to create a list and then go ahead and change elements to different values within that list. But now that we have a list object that we can mutate, other operations we can do with it is to, let's say, add more items to the end of the list. So we can make the list bigger.

We can mutate the object by doing that using this append function. Now I'm going to talk about the syntax of the append function in a little bit. But basically, if I want to mutate L, to add an item to the end of it, I have to use the syntax. There isn't a different form, a different function to do this. So this specific syntax has to be used.

Append is basically the function name, element is going to be the parameter, the thing that I want to add on to the end of my list, and L, the thing before the dot, is going to be the object I want to add the element to the end of the list. So L, in this case, I'm using it generically. But you can imagine creating a list of employees in your company. Then you might name that list employees. In that case, we would say employees.append(Anna), or whatever.

So that L is just kind of generic for now, but it gets replaced with whatever variable name your list is. So this operation basically mutates the list. So it mutates it to be one extra element longer. And the element you're adding to the end of the list, to the right-hand side of the list, is going to be whatever is in the parentheses to append.

So let's look at an example. So we're going to create L is equal to 213 in memory. And then let's say we do L.append(5).

Well, this line of code says, look up L. It's this object in memory here, 2, 1, 3, and add the object 5 to the end of it. So I'm going to add the 5 to the end of the list.

Now it's no longer three elements long, it's four elements long. And again, I didn't make a copy. I didn't preserve the original list with just 2, 1, 3 in it. I have literally changed this list in memory that's referenced by L.

Now this function append is being used basically for its side effect. And the side effect here is mutating the list. After the function adds the 5 in this particular case to the end of the list, the function doesn't need to return anything back. It's basically done its job to do the mutation.

And so functions like append, and we're going to see other functions later, don't have any return value. So one really common mistake, as we're learning about mutable excuse objects and using these functions that mutate, is to say, well, I'm going to do L.append(5) and save this, the result of this function, back into the variable named L. And this would be incorrect.

So let's see if we do this line of code, what exactly will happen. So it's an assignment. So the first thing we do is we look at the right-hand side and we evaluate that.

Well, the right-hand side basically says L.append(5), which is exactly the same as the previous line. So we're going to put another 5 to the end of our currently mutated list. Just going with these operations in order.

And then I said this append function has done its job to mutate the list by adding a 5 to the end of it. So it returns nothing. There's nothing of value that it could return, because it already did its job of mutation. So it actually returns none.

So the assignment, the equal sign, then basically says take the name L and bind it to the return of this function, L.append(5). Well, the return of the function is none, so basically now we're losing the binding from 2, 1, 3, 5, 5, which was our mutated list, and rebinding it to the return none. So that is an incorrect way to do the mutation of adding an item to the end of the list.

Everyone OK with that so far? Yes. OK, excellent.

So what should we have done instead? Sorry, yes, be careful about the append operation. You're doing a mutation and you return none as a result. So you do not want to resave this to any variable. So instead what we would do is we would just do the operation.

There's nothing to save. Nothing to save in any return variable. So if you wanted to add two 5's to the end of that list, you would just say L.append(5) again and L would then have been mutated to be 2, 1, 3, 5, 5.

And so in your code, if you just print L in between these appends-- the if you printed L after the first append(5), it would print this 2, 1, 3, 5. And then if we print L after the second append 5, it would print 2, 1, 3 5, 5, because it's an ongoing operation. It's mutating this list, and now you're doing operations on the newly mutated list. Everyone-- yeah?

AUDIENCE: For the element, can you only use one integer, or can we do 5, comma, 5?

ANA BELL: For the element, do you have to do one integer, or can use 5, comma, 5? So you can-- the append only works with one thing. So if you wanted to append a tuple, you could append one tuple object that has many things in it. But it would just append that one tuple.

We're going to see towards the end of this lecture an operation that allows us to extend the list by a bunch of items. But there is a way, just not with append. Yeah?

AUDIENCE: Why did you [INAUDIBLE] again?

ANA BELL: So the other thing-- so this operation always returns none. L.append(5) or whatever, the append always returns none. But here, it's just sitting on a line by itself. We're not saving it back to anything. In the previous one, we took the return and saved it back into L, and that's why we lost the binding to the actual list.

So what we usually say is that we use append and a bunch of these other mutable functions for their side effects. And the side effect in this case is to mutate the object that I'm calling the append on. In this case, the list named L.

So let's have you think about this problem and while you do it-- and then we can write it on the board together. So as we go through these lines of code one at a time, what will the values of the lists become?

So L1 is the string "re," L2 is "me," L3 is "do." What is L4 going to be right with that line L4 equals L1 plus L2. Does anyone know? What's the type? Its concatenation, right? So concatenation with lists is like concatenation with strings. Yes?

AUDIENCE: [INAUDIBLE]

ANA BELL: Yep. What are the elements in it?

AUDIENCE: "Re."

ANA BELL: Yep.

AUDIENCE: And then "mi."

ANA BELL: Yep, exactly. I'm not going to do the strings, but you know what I mean. So L4 with that line is just these two elements in a new list. Now what happens with the next line, L3.append(L4)? Which one gets mutated, L3 or L4?

AUDIENCE: L3 gets mutated.

ANA BELL: L3 gets mutated, exactly. And what does it get mutated to? So L3 originally has "do" in it. What am I adding to the end of L3?

AUDIENCE: [INAUDIBLE]

ANA BELL: Exactly. Yes, I'm adding one item, and it's linked to the question that was here. What am I appending? I'm appending one item. It's whatever L4 is.

And L4 is this list. So I'm going to be adding "re" and "mi" within my list here. And I've got to close this list here. So this is one item, one object, one element, and this is another element right here. It just happens to be a list.

What about the next line, L equals L1.append(3)? What is the right-hand side going to give me? Am I mutating L1 or L3?

AUDIENCE: L1.

ANA BELL: Yes. And what am I mutating L1 to be? L1 is originally "re." And what am I adding to the end of it?

AUDIENCE: [INAUDIBLE]

ANA BELL: Yeah, exactly. This L3, which is this big thing here. So it's a list with two elements, the first one being a string and the second one being another list like that.

So that's the right-hand side. And then what is the left-hand side going to be. What is L going to be? None, exactly. Yeah. Exactly.

So now that we've introduced mutable objects, we have to be careful about what functions we're using. Some of them mutate the list and don't return anything. Append is one of them, and we're going to see a few more in today's lecture.

So these functions are just being used for their side effect. They mutate the thing you're calling the function on and that's it. They don't need to return-- they don't return anything they, don't need to return anything. They have done their job purely by the mutation aspect of it.

So I want to just quickly make an aside on this dot notation that we've introduced with this append function. This is something we haven't actually seen before, but it's something that we will learn about in the future when we create our own object types. So right now, we're using object types that somebody else wrote, like a list, or a tuple, or something like that. But in a future class, we're going to learn how to create our own object types.

And when we do, we're going to use this dot notation a lot. But for now, you basically just kind of have to remember which functions use dot notation and which don't. But I'll give you a little bit of intuition for what this dot notation actually means.

So when we have-- so everything in Python is an object. And when we have objects in Python, the idea here is that the objects that you have data associated with them. So what makes up the object.

And they have certain behaviors. So we touched upon this on maybe the first lecture where we said things you can do with integers are different than the things you can do with strings. That's pretty clear.

And that's different than the things you can do with lists. And so the kinds of things that you could do with each one of these object types differs depending on the type. And at its core, really, everything can be written in terms of this dot notation. But some of the more common operations, like getting the length of something or adding two numbers together, are actually-- we do them in this shorthand notation, like using the plus operator or using the L.

But at their core, really, we can take all of those operations and convert them to a dot notation. We're not doing this today, but that's what we can do. And so when we see this dot notation, the way we usually read it is we say, well, what's to the left of the dot? It's going to be our object, the thing that we want to do an operation on.

In this particular case, it's a list named L, but it could be a list name employees, or words, or whatever. Book, or whatever the list contains-- whatever the list name is. The dot then comes for the dot notation and then the thing on the right-hand side is going to be the operation that you want to perform on the object to the left of the dot.

So the operation-- if you basically cover up L dot, the operation looks just like a function. It's append, parentheses, some parameters. And so the operation is basically just a function that you want to run on an object of type list, this specific object named L. And you can see it has a name, append, and it has parameters or arguments. In this case, it's the thing you want to add to the end of the list.

So again, unfortunately, at this point in the class, you just have to remember which functions are dot notation and which ones are not. But it will become clear what this dot notation actually means towards the end of the class.

So let's have you work on this little code here. It's going to use append, obviously, and it's going to have you create a list. So the name of the function you should make here is called--

[AUDIO OUT]