[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 this. 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 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?

All right, 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. So I'm basically binding the name L to that object in memory.

L square 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 5. 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 kind of make a version of that object. We have changed the object itself.

So let's see maybe similar code that you might think does the same thing, except with tuples. All right. So the first two lines of code are going to be the same. We've got L in memory being the object 2, 4, 3. 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 we 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 Ana 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 2, 1, 3 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 and 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 kind of 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 kind of going with these operations in order.

And then I said this function, 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 fives 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, if you print it L after the first L.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.

STUDENT: For the element, can you only use one integer or can you do [INAUDIBLE]?

ANA BELL: For the element, do you have to do one integer or can you use 5 comma 5? So 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?

STUDENT: Why did the other thing return none 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 mi. L3 is do. What is L4 going to be with that line L4 equals L1 plus L2? Does anyone know? What's the type? It's concatenation. So concatenation with lists is like concatenation with strings? Yes?

STUDENT: [INAUDIBLE]

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

STUDENT: [INAUDIBLE]

ANA BELL: Yep, exactly. I'm not going to do the strings, but you know what I mean. All right. 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? 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?

STUDENT: [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?

STUDENT: 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?

STUDENT: [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? Yeah, exactly. Exactly.

OK, 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 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 len. 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 named employees or words or whatever, book or whatever, 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 make ordered list. And it takes in one parameter, an integer n. It's positive. And I want you to create for me a list that has all of the integers from 0 all the way up to and including n inside that list in order.

So as an example, down in here around 34. If we call make ordered list with 6, it's going to create for us this list inside the function and return this list. So a couple of minutes to work on that and then we can write it together.

All right. What's the first thing we should do here? Or how would you approach this problem? Yes. You want to create an empty list. What do you want to name it? You named it list. List is an OK name, but notice list is also the name of the type of the object. So I would refrain from naming anything things that change color. So we can use L or my list or whatever. Something else. My list is an empty list. All right, so it's originally empty, and now we need to populate it with some stuff. Do you want to go on?

STUDENT: Make a for loop.

ANA BELL: Make a for loop. That goes over what?

STUDENT: [INAUDIBLE]

ANA BELL: Yup. 0 to n plus 1. Exactly. Because we need our boundary to go up to and including n. Perfect. So now that I've got I changing to be 0, then 1, then 2, then 3, what do I need to do to my list? Yeah, exactly. Append i. So my list is the name of the list I've created dot append I. The last thing, return the list. So return my list. So run it. Perfect. If we change this to 2, still works. So just testing it out with a couple different inputs just to make sure it works. Questions about this code? Yeah.

STUDENT: [INAUDIBLE]

ANA BELL: The 0 is not necessary in the range. It defaults to 0. Exactly. We're not done yet. You have more writing to do. So let's write a slightly different function now called remove LM. It takes in two parameters. The first one is a list, and the second one is going to be just some variable. It could be an integer, it could be a string, it could be whatever. And what the function should do is create a new list, populate it with the same elements of L in the same order, but exclude the ones that are equal to e. So you don't want to include the ones that are equal to e. Otherwise, keep everything in the original list L in the same order.

So as an example here, we've got if our input list is 1, 2, 2, 2, 2, and I call the function with L and 2, the list that this function returns should just contain one element in it. Just the one. So try your code for the next couple of minutes around line 50. And then we can write it together. All right. How can we start? Yes.

STUDENT: I created a new list.

ANA BELL: Yep. What did you name it?

STUDENT: [INAUDIBLE]

ANA BELL: Cool. What did you make it be-- empty list. OK.

STUDENT: And then I said [INAUDIBLE].

ANA BELL: Yep. So for I in L. And at this point, I would make a note for myself, because you use I, which in my brain means index. But I would make a note for myself that I is maybe 1, then 2, then 2, then 2, just according to this first example. So if I'm reading the code, I would just-- I will remember that it's not the index, but go on. So for I and L directly.

STUDENT: [INAUDIBLE]

ANA BELL: Like this? OK.

STUDENT: [INAUDIBLE]

ANA BELL: Yeah. OK. And then that's good. Return your list. OK. Let's try it.

STUDENT: [INAUDIBLE]

ANA BELL: Yeah, so e is not a list. e is going to be an element. So that's my bad. I should have put this in here. e is an object or something. It could be a list, but then I would be looking for that exact sublist, that exact list as a subelement. So maybe we think of e as an object like 5 or something like that.

STUDENT: [INAUDIBLE]

ANA BELL: Yeah. Not equal to e. So if I'm just looking for that element directly, I want I to be not equal to e, in which case I keep the element in my new list. So if we run that, that gives me 1 according to this. Looks like it's correct. And then we can run it with these other two cases. So here I'm removing the element 1. So I'm going to keep 2, 2, 2 as my returned list. And here I'm removing 0, which doesn't exist in my list at all. So it should just keep, and it does, my original list unchanged. Any questions about this example? Anyone try it a different way? OK.

All right. So other useful list operations. We can convert strings to lists and then lists back to strings. And this is very useful when you're reading in text or something like that to a function. It's going to be useful for problem set 3, so on and so on. So let's first see how we can take a string s and convert it to a list.

So if we just cast s to a list, the way we used to cast the number 5 to a float, we would just say float parentheses 5. Well, we can take a list and cast it to a list by saying list parentheses s. And if we cast it like this, Python takes every single character in s and makes it be a separate element in a list. So you can see here I've got the string I heart cs &u. It makes for me a list where every single character, including the space and all the special characters, becomes a separate entry in my list.

That's not that useful. I mean, it can be, but it's not that useful. What is more useful is to take an input string and split it on a particular character. So one very common character that we would split on is the space. And if we do something like that, it basically extracts from us, from our string all of the individual words, which is pretty useful. So here I've got s.split. And in parentheses, I've got the character I want to split on. In this particular case, a space.

So if I take s and I split on the space, Python will go from the beginning of the list to the first space. Make that be one element in the list. It'll go from the first space to the next space in my string and make that be the next element in the list. And so on and so on, until it gets to the end of the list and makes that last bit the last element in my list. So here, when I've split on the space, I've got 3 base words, quote unquote "words." I heart is going to be one. And there it is as my first entry. CS is in between these two spaces and that's my next entry. And, &u? Is my last entry here.

So this is a very useful function. We can, of course, split on any character we'd like. So here I am, splitting on the less than character. So there's only one. So if I split on the less than character, one element in my resulting list is just the capital I. And the remaining element in my resulting list is the 3 space cs &u. And there it is right there.

All right. So once we have a list, we can also go backward. We can take this list and convert it back to strings. So we use this join function here. And the thing before the dot is going to be what character you want to join the list elements with. And this is the list you want to join back into a string. So let's look at an example.

So let's say I have list L that has three entries in it, A, B, and C. If I join on the empty string, so here this is just quote quote beside each other. There's no space or anything in there. That's going to take from me all the elements in the list L and join them together as one. Nothing in between the A, B, and C. And this operation here will basically make for me A, B, the string A, B, C.

If I join on an underscore, you might have guessed, it'll join A, B, and C with an underscore in between each character. So there it is. A_B_C. You can join on any character you'd like. I don't know if you can join on multiple characters, but I don't see why not. You could try this out on your own.

Join only works with lists that contain only string elements. So if we try to join a list that has just integers or floats or Booleans, anything that doesn't contain a string in it, then you will get an error. Because it's basically trying to put all these back into a big string. If you wanted to join non-string elements, you would have to basically loop through and cast every one of these to a string first and then join them together. So if you want to join 1, 2, 3, you would have cast them to strings and then you could join them to make the string 1, 2, 3.

OK, so let's have you work on this example. So here we're going to try to split the input. So here is a function called count words. It takes in one input sen for sentence. And I wanted to use something that's not s just to make it clear that the thing before the dot isn't always s. It's whatever object you want to split or join or whatever. So this function is going to return how many words are in s.

Quote unquote "words" in this case, because I'm just interested in the elements or the characters between spaces and between the start and the end of a word. So if it's a number, I still count that as a word. If it's a special character, a dot, exclamation point, I would still count that as a word as well. So this should be just a couple of lines of code down around 99. So I'll give you about a minute to work on it, and then we can write it together.

So thoughts on how we can do this? L1 equals sen.split. Yep. Sorry, parentheses, space. Yep.

STUDENT: [INAUDIBLE]

ANA BELL: Yep. And then we can return the length of L1. Perfect. Let's run it on these two examples. And should print 3 and 12. And it does. So notice how easy this was with lists, because lists are data structure that's just kind of naturally iterative. And so running len on this split list or split string, which gave us a list, is really easy. It's a two line piece of code.

Without lists, you could imagine creating variables that keep track of where you see the first space and then iterating through one character at a time. And if it's a space, keep track of the fact that you saw a space and then look for the next space and then resetting things every time you see a space. And that would be really, really tedious. It would be a really good quiz one question, but not once we've introduced lists, because it becomes really, really easy to do it with lists.

All right. So now that we have lists, we can do other really interesting and useful operations to mutate the list. So we saw the dot notation on a list to do append. So basically, to add an item to the end of our list. That was useful. Other things we can do in terms of mutating the list is to sort a list and reverse a list. And these are also very useful operations on lists. So the first two here, sort and reverse, is the notation for how we sort a list and how we reverse a list. And these will mutate the list that you call the functions on.

So if I have list 4, 2, 7 here and I call L.sort and I print L as the next line after this, L will have changed in memory to be 2, 4, and 7 in that order. It didn't make a copy for me. It didn't preserve the original order. It changed that list to be now in sorted order.

Reverse similarly. So if we do L.reverse on 4, 2, 7, it will reverse all the elements. So the one at the end becomes at the beginning. The one second last is the second one. Third last is the third element in the list, and so on. And again, this mutates my list. So I would have lost my original order with this command, L.reverse and with L.sort, of course.

Now, there are many situations where you want to preserve the original order. I don't know, maybe like the order that people join a company or the order that people joined a grocery queue. I don't know, things like that. You might want to preserve that original order, but you might also get maybe the sorted names of people for a function that does something with those sorted names.

In that case, you don't want to call sort on your original list, because you would lose the original order. You could, of course, make a copy or you could call this sorted function. And the sorted function is going to keep my original list L intact in the same order that I had created it in. But it would return for me, so this function will actually make a copy and return for me the sorted version of L. And L remains unchanged. So this function does not do any mutation. We have to take the return and save it into a new variable. This case I called it L new.

So might be a little bit sort of hard to keep straight in your mind, whether to use sort or sorted. You could, of course, always try it in the console to see which one does what. The way I remember and think about it is the sort to me feels like a command. It's like sort this list. Mutate this list and sort it. Whereas sorted is more of a request. Can you please get me the sorted version of L? And so that's kind of how I keep things in my mind as to whether I'm calling sort to do the mutation or asking to get the sorted version of the list. Yes?

STUDENT: What is it sorting by?

ANA BELL: It is sorting it by whatever the built in sort is for those particular object types. So in the case of integers, it's just increasing order. In the case of strings, it'll be alphabetical. You can choose different sorting functions, but we don't get into that. Yeah.

STUDENT: [INAUDIBLE]

ANA BELL: That's a good question. I think they do in order for it to work. So we can try L equals 1 and then we can give it a tuple or something. And then we can ask sort L. Yeah. So in this case, it doesn't know how to resolve. It's trying to do a behind the scenes less than to figure out which one is bigger than which.

And in this case, it doesn't know how to resolve. How do you choose whether the tuple is bigger than an integer? But you can imagine, again, as I mentioned, this is not something we do, but you could write your own sorting function where depending on the type, you would decide which one is bigger. So yes. Question?

STUDENT: [INAUDIBLE]

ANA BELL: So you would just do L.sort without parentheses, but L has to be a list that contains things that can't be sorted. So all integers, all strings, or something like that.

So let's look at the memory diagram for how this would look, just to bring the point home about objects that are being mutated. So our original L is 9, 6, 0, 3. So in memory, I've got the name L bound to 9, 6, 0, 3. Again, let's do an append just for fun. L.append 5 is going to add a 5 to the end of that list. And append, sort, and reverse will all be used for a side effect. That means they're going to be mutating the object, whereas sorted will not do a mutation.

So let's do an append to the end. That's going to put a 5 at the end of the list. Something we already know how it works. Now let's do A equals sorted L. So again, it's an equality. So the thing on the right hand side is going to be the function that returns for me the sorted version of L. So it's going to create a new object.

However it does the sort, it's going to create for me a new list that contains that sorted order. The original L notice in memory remains unchanged. So if I want to reference L in my program from here on out, it will use this unchanged L. So now the return of sorted is this list and I bind it to A. So name A now points to the sorted list version.

All right. Now, what if I do this line here, B equals L.sort. Again, let's look at the right hand side. L.sort is going to mutate L. So this function itself will go and change L to be-- L's object, the object that L points to, to be the sorted list. But it's not done. This function is being used for a side effect. So what is the return from it? None, right? It's like the append. So this example here will make B point to the return of that function, which is just none.

Now, please don't ever do this. All you would have to do to sort L is to just on a line by itself say L.sort. I just did this to show you again that if you do L equals L.sort, bad things will happen. You're going to reassign L to be none. In this case, I saved it under a different variable, but it's an easy mistake to make.

And then what about the last one here? L.reverse. Again, I'm going to go and grab the object pointed to by L, and I'm going to reverse all the elements. So here doing L.sort and then L.reverse right afterward makes my list be in reverse sorted order. So biggest number to smallest number. So with that command there, I've got 9, 6, 5, 3, 0 instead of 0, 3, 5, 6, 9. And again, sort and reverse changed my list L directly. So I've lost that initial order of 9, 6, 0, 3 that I had up here.

One last point I want to make. I know we've usually seen functions that take in parameters. Sort and reverse are still functions. And they just happen to not need any parameters. You call them on the object L using this dot notation. So in effect, it does have sort of a quote unquote "parameter," the thing before the dot. But it doesn't take anything else in their own respective parentheses. But they do still need the parentheses there, because they are functions. They are operations that will do something for us. Questions about this? Is it OK?

OK, very good, because now you get a chance to try it out. So let's have you do something similar to what we did last time. Take in a parameter sen, which is a string representing a sentence. I want you to figure out all the words, quote unquote, in the same manner that we did before in the previous example, but now return for me a list with these words in sorted order. So if the input was look at this photograph as my sentence, then I would return a list which has at look photograph and this as my four elements in that order. So here start writing it down on line 134.

OK, what is the solution? What do you have so far? Yes.

STUDENT: L equals sen.split with the [INAUDIBLE].

ANA BELL: L equals sen.split and we split on a space. Got it. OK.

STUDENT: And then L.sort parentheses with nothing around.

ANA BELL: Got it. Return L. Perfect. OK, let's see if that worked with our two examples. Yep. There's my first one. There's my second one. Anybody do it a different way? Did anyone use sorted? Yeah.

STUDENT: I just said return sorted parentheses L.

ANA BELL: Return sorted parentheses L. Yeah. Is that how you--

STUDENT: [INAUDIBLE]

ANA BELL: Yep. We could do it all in one. Perfect. Yeah, this could be a one liner, for sure. Yeah, so this works because this thing here creates for me a new object. I could have saved it in a different variable and then returned that variable, but this does it all in one. So just for completion sake, if we comment out the other solution, this way still works. Questions so far? OK.

All right. So what we've seen so far is a bunch of these functions, built in functions, that have these side effects. They mutate the input list. So we can actually write our own functions that have a side effect, where if we pass in a parameter that's a list, we can have our functions mutate that list however we'd like.

So let's go through this example. Let's say we were given the task of writing a function that takes an input list L and mutates the list L such that each element in L is changed to be the element square. So 2, 3, and 4 as an input list becomes 4, 9, 16. And I'm mutating that list. I'm not creating a new list and returning the new list. I want to actually mutate the input list L.

So if we were faced with this task, the way that we would go about it, maybe based on what we've learned so far, is to say, well, I'm going to iterate through each element in L because that's a very Pythonic way to do this. So I'm grabbing the element in the list L. But then I would be stuck. Because the syntax for changing an element at a particular location is L at I equals whatever the changed thing is. But my loop variable is iterating through the element directly. So what's my index in this particular case? I don't have it in hand. I have the element, but I don't have the index.

So what are some solutions? Well, a first solution could be right before the loop to create a new variable that keeps track of the index. So you make I equals 0 right before the for loop. And inside the for loop, you increment I each time. Now you're keeping track of the index yourself.

Option two is to change what we iterate over. So instead of iterating through each element in L directly, let's iterate over the index. So iterate over range length L. In that case, the range length L basically becomes range 5 or 20 or whatever the length of my list is.

And a last option is to try to use this thing called enumerate, which is a Python keyword, I guess, Python function. And the syntax for that would be for tuple I comma e in enumerate L. So I'm basically wrapping L inside this enumerate function. And Python each time through the loop makes this little tuple I comma e be the index and the element at each location each time through the loop. And so it gives you a two for one kind of deal here using this enumerate function.

I'm not going to go over option one or option three. I do encourage you to try to look these up or try to implement them yourself. But I will go over option two in these slides. So if I were to iterate over the index directly, the way I do it is I'd have to change the loop variable for I in. And then the thing I want to loop over is all of the indices. So I want to get the numbers 0, 1, 2, 3, 4, all the way up to the but not including the length of L.

So once I have this index in hand, I can do something like this very easily, because this I here is going to be my index. And the thing on the right hand side is just going to be a matter of grabbing the element at that index and squaring it. So here L square brackets I on the right hand side grabs for me the element at index I.

So what's the value of that element at that particular location? 23 whatever. Square it. So **2 squares it. And then the thing on the left hand side is the syntax for changing the element at a particular location. We saw this way back on slide two. So with this line of code, Python goes through each element in the list and squares it and saves it back into that same list. No new list is created. It's mutating the original list. No return. Nothing to return. This function will return none because it does its job of doing the mutation.

So if we go through an example, suppose that L is 2, 3, 4. What is this loop going to do? So I the first time through the loop will be 0. And then that first time through the loop, it will mutate I. So it says L square bracket 0 equals whatever the element at 0 is, the 2 squared. So L square bracket 0 will be changed to 4. So we've mutated the element at index 0 to be a 4, and everything else is the same.

Next time through the loop, I'm mutating the list that I had just mutated. So the first element is still the mutated value 4. But now I'm going to change my element at index 1 to be 3 squared 9. The last time through the loop, all the elements up to index 2 are going to be the mutated elements. So 4 and 9. And then the last time through the loop by mutating the 4 to be 16, the square version of that.

So to check that we did the mutation correctly, what we would do is we would create an input list. I called it L in. And I've set it to 2, 3, 4, my example. If I print before the function, call the value of L in, it's 2 comma 3 comma 4, as expected. It shouldn't be anything different than that. Then I make a function call to the function that we just wrote. Note I'm not returning anything here. So I'm not saving the function call to any variable.

If I print L in after the function call, it will print the mutated list. So this L in here and here and here is the same object. Nothing was returned. This function here has nothing to assign its return to. If we assigned it to something, that variable would be none. Just like the append, just like the sort, just like the reverse. All right. So when you're writing-- oh yeah, question.

STUDENT: So we created a function, but we don't have a return and it doesn't say none.

ANA BELL: We created a function. It doesn't have a return. It doesn't say none because we didn't save the function call to any variable. If we said A equals this function call, if we print A, it would show none. Yeah.

OK, so when we're writing functions that mutate input lists, the two likely things you're going to have to do, and it depends on what your function is actually doing, but most likely you're going to have to iterate over the length of the list. So for I in range length L to grab the index as well as the element. To be able to grab the index as well as the element. And these functions, I mean, they could do other stuff. But if you're using them for mutation and things like that, they're going to return none. So when you make function calls to them, those function calls will likely just be on a line without saving the return to any variable.

So we've talked about mutable objects. They're very, very useful. Places where they're useful or the reason that they're useful is because they allow you to have basically large databases of objects like employees in a company list of all the students at MIT, things like that. And if you want to make a change to something about that list, like a student changes their name or their address or something like that, with tuples, you'd have to make a new copy of that entire list. So it could be very space inefficient, because every time a student changes their address or their name or something about themselves or their grades or something like that, you're making a new copy of this potentially thousands long data structure.

Lists don't have that issue. With lists, you're just mutating the object in place and you're done. No extra copies are being made, so it's a very efficient data structure. But with lists come some unexpected challenges. And we're going to go through three tricky examples today. In next lecture, we're going to see tricky example number four. And these three tricky examples involve looping over the list in one way or another. Over the range of the length of the list or through the list directly.

So let's look at the first example. In this code down here, we're going to loop over the range of the length of the L. And then what we're going to do is append the loop variable I to the end of my list. Now, what does range length L do? So remember the thing that our for loop iterates over is a sequence of values.

Now, range some number creates for us a tuple like object. Not a tuple specifically, but you can think of it like a tuple. So range 4, the length of this particular list, would create for us in memory something like a tuple. The sequence 0, 1, 2, 3. And this is the sequence that the loop variable I will go over. First it will be 0, then it will be 1, then it will be 2, then it'll be 3.

So when we iterate through the sequence, Python says, OK, the first time I encountered this for loop, I'm going to save this sequence I need to iterate over as an object in memory. And then I'm going to have my loop variable iterate over each one of these elements. The thing I'm doing is appending I to the end of a list. So the first time through the list I'm going to append a 0 to the end. So the 0 being this loop variable here. Next time through the list, I'm appending the 1 to the list I just mutated. Next time, I'm appending the 2 to the list I just mutated.

And the last time through the list, I'm appending the 3 to the list I just mutated. And we finish. We've gone through four times. We've appended four items to the end of the list. 0, 1, 2, and 3, the elements of my sequence that I'm iterating over.

Let's look at the memory diagram. So originally L is 1, 2, 3, 4. What exactly happens when we first encounter range length L? That gets put as a variable, this tuple like thing. I made it be a tuple, but it's not exactly a tuple in memory. And this I will iterate through each one of these values in my sequence. This is the sequence of values that I'm going to iterate over.

So the first time through the loop, Python has I pointing to 0 here. And so what's it doing inside the loop? It's going to append the 0 to the end of L. Next time through, then it's going to print L. Sorry. And then next time through the loop, the loop variable increments by 1. So we've already looked at the 0. Now we're going to do the 1. So the loop variable I is now 1. So we're going to append the loop variable 1 to the end of L. Print L. Loop variable becomes 2. Append the loop variable to the end of L. So now it has a 2. Print L. And then the last time we append the loop variable 3 to the end of the list L and print L.

Pretty straightforward. The code terminates because we've created this original tuple-like object here, which tells Python what values you need to iterate over. This is your sequence of values to go through. So that's basically what I said.

So let's look at a slightly different example. So in this case, instead of iterating over the range length L, let's iterate over the elements in L directly. So for e in L. Now, to keep things in parallel to what we had done before, let's create a loop variable I equals 0 before the for loop and let's increment it by 1 each time through the loop. So we're going to still append 0, then 1, then 2, then 3 to the end of our list.

So in this particular case, we start out with the memory diagram like this. So we have L pointing to 1, 2, 3, 4. Loop variable I is going to be 0 originally. And e will first point to the first element in the list. That's what for loop over the elements in the list does.

So going into the list, we say L.append I. So at the end of L, I'm going to mutate it to contain a 0. Good? I increment I by 1. Good. I print L OK, good. And then the next time through the loop, Python says, all right, what's the next element in my sequence? Well, I looked at the 1 first. Now let me look at the 2.

All right. Now I'm looking at the 2 as my next element in sequence. I'm going to append I to the end of the list. So I'm going to append 1 to the end of L. Increment I by 1 to be 2. Print L. OK, next value in my sequence. e increments to the next element in the sequence, the 3. We append I to the end of the list. So we append 2 to the end of L. Increment I by 1. Print L.

What do you notice? Is this code going to terminate? No, because our loop variable will always be four elements away from the list, the end of the list. As I'm adding an item to the end of my list, the loop variable iterates to the next item. But then I'm adding another item to the end of my list, and my loop variable iterates. And so we're always going to be 4 behind the end of the list. So this code will actually never stop.

All right. So the difference here is what I'm iterating over. In the previous example, as soon as Python saw range, length, whatever, it made this predefined sequence of values it needed to iterate over. But here it doesn't do that, because it's iterating over my object L. There's no predefined sequence to create. It's supposed to iterate over L directly. So that's the difference between these two.

All right. So now I'm going to show you-- before I do the last trick example involving concatenation, I wanted to mention one thing, which is there was a question earlier, how do we actually add more than one item to the end of our list? And we do that using this extend operation. And this extend operation is kind of like append, but we are going to add all of the elements of some list as the parameter to the end of our list L. So in effect, we're mutating L to be extended by all the elements in some underscore list.

So here's an example. First, let's do concatenation just to remind ourselves what it does. So L1 is 2, 1, 3 in memory. L2 is 4, 5, 6 in memory. L3 is going to be L1 concatenated with L2. Pretty straightforward. It's concatenation. So Python creates for me a new object, which is all the objects in L1 and L2 put together as this completely new object bound to the name L3. So L1 and L2 remain unchanged. No problems there.

But the extend is going to mutate. Notice the dot notation format of extend. It's going to mutate L1 to be extended by all the elements in this list. So it's going to add a 0 and a 6 to the end of L1. So here it is. I've got L1 mutated to be 2, 1, 3, and then 0 and then 6.

So just to bring the point home, the thing we're extending by is all of the elements of this list in the parameter. So in this particular case, L2.extend will be extended by how many elements? 2 or 4? Yeah, I see 2. Exactly. It'll be extended by 2 elements. At the top level, this list has two elements in it, a list and then another list. So this command here will extend L2 by these two elements specifically, 1 comma 2 as a list and 3 comma 4 as a list. But these are individual objects. They're single objects that are lists. They happen to have a bunch of elements as part of the list, but they are two objects. Yeah.

STUDENT: [INAUDIBLE]

ANA BELL: When we extended by 0 comma 6, there's no brackets because we're extending it by the elements of this top level list. So it's two integers. And here we're extending it by the elements of this top level list. So in a sense, the outermost parentheses, which they are lists. Yeah.

So that introduces extent. We're not actually going to use extent for this particular example, but I did want to mention it. In this example, we're going to use the concatenation operator to create for us this new object and bind it to L again.

So let's see what this is going to do. First, I'm going to actually tell you the answer, and then we'll do the memory diagram again to bring the point home. So this loop will, again, loop through all the elements in L. Originally it's 1, 2, 3, 4 in a list.

So what is the actual loop going to do? It's going to take whatever is in L, double it. So originally L is 1, 2, 3, 4. The first time through the loop is going to create a new object, which is just L 1, 2, 3, 4 doubled. I've concatenated L with itself. And then I'm going to save it as this new object with the name L again. That's the first time through the loop.

Second time through the loop, I'm going to take whatever L was mutated-- or whatever L was before. Sorry, not mutated, but whatever L was before. So it's 1, 2, 3, 4, 1, 2, 3, 4. Double that and save it under the name L. Second time through the loop. Now, third time through the loop, I'm going to take whatever L is right now. So these two rows of 1, 2, 3, 4, 1, 2, 3, 4. Double that and save it under the name L.

And then the last time through the loop, I'm going to take whatever L was before. So these four rows of 1, 2, 3, 4, 1, 2, 3, 4, double those, and save that as the new L. And that's it. This code does not go to infinity.

Now, let's see why exactly that is. So this will help. Originally, I've got L is 1, 2, 3, 4. So that's straightforward. My loop variable e goes through each element in this object. So first it's going to point to the 1. So far the same. L equals L plus L. Let's look at the right hand side first. This creates for me a new object. Remember, concatenation creates for me a new object. It doesn't mutate anything. So in memory, I'm going to get 1, 2, 3, 4, 1, 2, 3, 4. I've doubled L with itself.

What is L equals going to do? Do you remember we did a slide like this, very similar to this. It's like when we reassigned the tuple. The L equals will actually take the binding from my original object and put it on the new object that I just created. Exactly that memory diagram with the tuple. I happened to have the same name, but it's pointing to a new object. Same here. I happen to have the same name, L, but it's now pointing to this new object. The old object, this thing that I'm iterating over, I've lost the binding to it. And really the only way I can even reference that old object is through this e, because that e is still going to go through this old object element. So that's the first time. Yes, question.

STUDENT: Why does e not [INAUDIBLE]? Because you defined it before you [INAUDIBLE]?

ANA BELL: Yes, you define it to be the object in memory, not the name L. So e is bound to the object in memory. That's this thing here. That's why it was so important to separate ourselves from the object in memory versus the name we give an object, because that name can change to anything, to a bunch of other stuff. But the object itself remains in memory.

So then this becomes pretty straightforward if you understand that piece. The first time through the loop, I've got L assigned to this new object here. I've lost the binding to my original list that I'm iterating over. So when I print L, I print this. Next time e increases to the next element in my sequence. L will double what it currently is. So it's currently this thing here. It's going to double to that.

And I'm going to lose the binding from the original, or not the original, but this thing that I had just bounded to, to bind it to the next object that I just created. And then that's the second time through the loop. It looks like this. Third time through the loop. e increments by 1, to the next value in my sequence. I'm going to take L plus L, so double that previous data object. Take the binding from that previous object to the new one. Increment e by one more.

And this will be the last time E is going to change, because after this, e will have gone through all the elements in its sequence. It's gone through the end of that list. So the last time through the loop, I've doubled that L, and I've lost the binding from the previous one, made it to the new one and then that's it. It's done. Questions? Is it straightforward? Does the picture help all that? OK.

One more thing I want to mention, and this is kind of a preview of what we're going to do next time. One very useful operation that you might want to do is to take a list and remove all of its elements. Yeah, go ahead.

STUDENT: Sorry. [INAUDIBLE] because usually it stops before the last one.

ANA BELL: Oh, here that's in range. So here it's just iterating through all the elements in the sequence going to every one. So one useful operation we might want to do is to remove all the elements in a list, but not-- sorry, but by mutating the list. So we want to keep our original list object. We just want to basically clear it out of all of its elements. And so the command for that has a pretty nice name. It's called clear. So if you want to take a list L and clear it, so to remove all the elements inside it, you say L.clear. And that mutates my original list L to be empty.

So one thing that might help with this mutation lecture and figuring out which object is which and whether you've created a new object or not is to figure out-- is to ask, how do I know that this object is the object that I'm mutating? And to do that, we're actually going to use this function called ID. And ID lets us get the memory location or memory object or the idea of the object itself in memory.

So the code on the left is code that takes in a list L. We get its ID to see what is this object in memory. What's its number? Append an 8 to it. We're going to see that the ID of this is going to be the same as the ID of this. Because we're mutating L. We're not adding it. We're not changing it. We're not creating a new object.

And then lastly, we're going to clear it and check the ID again. And you're going to see that the ID is exactly the same in all of these different cases. So I'm doing this just in the console just to show you real quick. So here I have 4, 5, 6 as my L. Here's L 4, 5, 6. The ID of it is this number here. We can just look at the last three digits or whatever. 8, 0, 8.

Let's append an item to the end of our list. L mutated to contain that item. The idea of L remains the same. Ends in 8, 0, 8. It's the exact same object in memory. We've mutated it. L.clear. L empty list. I've removed all the elements of L. And the idea of L will show me that it's this exact same object. I'm just mutating this same object in memory.

Let's do that again, except in the new version instead of using L.clear, I will say L is equal to the empty list. And this is also a really common mistake to make. So here I have L is 4, 5, 6. Again, this is my L. Let's get the idea of L. It's going to be a new one, because I've reassigned L to this new list. So this one ends in 3, 1, 2.

Again, let's do an append just for fun. The ID or L ID of L is going to be, again, 3, 1, 2. But now if I say L is equal to the empty list, this is exactly the same as the situations we've seen before with the tuple and with that trick example number 3. When I say L is equal to the empty list, Python takes my name L and assigns it to this object that is the empty list. My original object, 4, 5, 6, 8 is still in memory. I've just lost the binding to it.

So here's L. It's an empty list. But the idea of L is now going to be different. Originally, I was working with this list at ID 312. But after I said L is equal to the empty list, I've lost the binding from that old list and rebound my name L to this new empty list. And you can see this using this ID, which is pretty cool.

OK, quick summary. So we saw lists and tuples as a way for us to create these compound data structures that can contain any kind of object as their elements. Tuples are immutable. So for things that don't change, they're very useful, like country, latitude, longitude. Those things won't change. Or the word that appears on a page number and a line number, something like that.

Lists are mutable objects, so you use them in situations where you need that dynamic aspect. So if you want to maintain a list of employees, you want to maintain a list of students, a list of grocery items or things in your fridge, those are really good situations where you'd want to list, because things are constantly changing. You don't want to make copies of everything all the time because it becomes very inefficient to do so.

So next lecture, we will continue with tricky examples. And we'll also have a quiz. Remember, quizzes are now on Mondays.