**PROFESSOR:** Quick, quick recap of what we did last time. So last time we introduced this idea of decomposition and abstraction. And we started putting that into our programs. And these were sort of high level concepts, and we achieved them using these concrete things called functions in our programs. And functions allowed us to create code that was coherent, that had some structure to it, and was reusable. OK.

And from now on in problem sets and in lectures, I'm going to be using functions a lot. So make sure that you understand how they work and all of those details. So today, we're going to introduce two new data types. And they're called compound data types, because they're actually data types that are made up of other data types, particularly ints, floats, Booleans, and strings. And actually not just these, but other data types as well. So that's why they're called compound data types.

So we're going to look at a new data type called a tuple and a new data type called a list. And then we're going to talk about these ideas that come about with-- specifically with lists.

All right. So let's go right into tuples. So if you recall strings, strings were sequences of characters. Tuples are going to be similar to strings in that they're going to be sequences of something, except that tuples aren't just sequences of characters, they can be sequences of anything. They're a collection of data where that data can be of any type.

So a tuple can contain elements that are integers, floats, strings, and so on. Tuples are immutable. And if you recall, we talked about this word a little bit when we talked about strings. So that means once you create a tuple object, you can't modify it. So when you created a string object, you were not allowed to modify it.

So the way we create tuples are with these open and close parentheses. This shouldn't be confused with a function, because there's nothing-- there's no-- this isn't a function call. It's just how you represent a tuple. For a function call, you'd have something right before the parentheses. This is just how we chose to represent a tuple. And just a plain open and close

parentheses represents an empty tuple. So it's of length 0. There's nothing in it.

You can create a tuple that contains some elements by separating each element with a comma. So in this case, this is a tuple that can be accessed with a variable t that contains three elements. The first is an integer, the second is a string, and the third is another integer. Much like strings, we can index into tuples to find out values at particular indices. So you read this as t at position 0. So the tuple represented by a variable t at position 0 will evaluate to 2, because again, we start counting from 0 in computer science. So that brings us-- gives us the first element.

Just like strings, we can concatenate tuples together. That just means add them together. So if we add these two tuples together, we just get back one larger tuple that's just those two-- the elements of those tuples just put together in one larger tuple. Again, much like strings, we can slice into tuples. So t sliced from index 1 until index 2. Remember, we go until this stop minus 1. So this only gives us one element inside the tuple.

And this is not a mistake. This extra comma here actually represents a tuple object. If I didn't have this comma here, then this would just be a string. The parentheses would just-- wouldn't really make any difference. But the comma here makes it clear to Python that this is a tuple with only one element in it.

We can slice even further to get a tuple with two elements. And we can do the usual operations like get the length of a tuple, which says, how many elements are in my tuple? And len of this t would evaluate to 3, because there are three elements inside the tuple. Each element, again, separated by the comma.

And just like strings, if we try to change a value inside the tuple-- in this case, I wanted to try to change the value of the second element to 4-- Python doesn't allow that, because tuples are immutable.

So why would we want to use tuples? Tuples are actually useful in a couple of different scenarios. So recall a few years ago, we looked at this code where we tried to swap the values of variables x and y. And this first code actually didn't work, because you're overwriting the value for x. So instead, what we ended up doing was creating this temporary variable where we stored the value of x, and then we overwrote it, and then we used the temporary variable. Well, turns out this three liner code right here can actually be written in one line using tuples.

So you say x comma y is equal to y comma x. And Python goes in and says, what's the value of y? And assigns it to x. And then what's the value of x? And assigns it to y.

Extending on that, we can actually use tuples to return more than one value from a function. So functions, you're only allowed to return one object. So you're not allowed to return more than one object. However, if we use a tuple object, and if that's the thing that we return, we can actually get around this sort of rule by putting in as many values as we want inside the tuple object. And then we can return as many values as we'd like.

So in this specific example, I'm trying to calculate the quotient and remainder when we divide x by a y. So this is a function definition here. And down here I'm calling the function with 4 and 5. So when I make the function call, 4 gets assigned to x and 5 gets assigned to y. So then q is going to be the integer division when x is divided by y. And this double slash just means-- it's like casting the result to an integer. It says divide it, keep the whole number part, and just delete everything else beyond the decimal point.

So when you divide 4 by 5, this q is actually going to be 0, because it's 0 point something, and I don't care about the point something. And then the remainder is just using the percent operator. So I divide 4 by 5, the remainder is going to be 4. And notice that I'm going to be returning q and r, which are these two values that I calculated inside my function. And I'm returning them in the context of this tuple. So I'm only returning one object, which is a tuple. It just so happens that I'm populating that object with a few different values.

So when the function returns here, this is going to say 0 comma 4. That's the tuple it's going to return. Q is going to be 0 and r is going to be 4. So then this line here-- quot, rem equals 0, 4-- is basically this-- it's sort of like what we did up here. So it assigns quot to 4-- sorry. Quot to 0 and rem to 4. So we can use tuples. This is very useful. We can use them to return more than one value from a function.

So tuples are great. Might seem a little bit confusing at first, but they're actually pretty useful, because they hold collections of data. So here, I wrote a function which I can apply to any set of data. And I'll explain what this function does, and then we can apply it to some data. And you can see that you can extract some very basic information from whatever set of data that you happen to collect.

So here's a function called get_data, and it does all of this stuff in here. And in the actual code associated with the lecture, I actually said what the condition on a tuple was. So it has to be a

tuple of a certain-- that looks a certain way. And this is the way it has to look.

So it's one tuple. The outer parentheses out here represent the fact that it's a tuple. And the elements of this tuple are actually other tuples. So the first element is a tuple object, the second element is a tuple object, and third one is a tuple object, and so on. And each one of these inner tuple objects are actually going to contain two elements, the first being an integer and the second being a string.

So that's sort of the precondition that this function assumes on a tuple before it can-- before it actually can work. All right. So given a tuple that looks like that, what's the function going to do? It's first creating two empty tuple. One is called nums and one is called words. And then there's a for loop. And notice here the for loop is going to iterate over every element inside the tuple. Remember in strings when we were able to use for loops that iterated over the characters directly as opposed to over the indices? Well, we're doing the same sort of thing here. Instead of iterating over the indices, we're going to iterate over the tuple object at each position.

So first time through the loop, t here is going to be this first tuple. The second time through the loop, t is going to be this tuple. And the third time, it's going to be this exact tuple object.

So each time through the loop, what I'm doing is I'm going to have this nums tuple that I'm going to keep adding to. And each time I'm going to create a new object and reassign it to this variable nums. And each time through the loop, I'm looking at what the previous value of nums was. So what was my previous tuple? And I'm going to add it with this singleton tuple. So it's a tuple of one character or one element. This element being t at position zero.

So you have to sort of wrap your mind around how this is working. So if t is going to be this tuple element right here, then t at position zero is going to be this blue bar here. So it represents the integer portion of the tuple.

So as we're going through the loop, this nums is going to get populated with all of the integers from every one of my tuple-- inside tuple objects. So that's basically what this line here is doing. At the same time, I'm also populating this words tuple. And the words tuple is a little bit different, because I'm not adding every single one of these string objects. So t at position one being the string part of the inner tuple. I'm actually adding the string part only if it's not already in my words list. So here, I'm essentially grabbing all of the unique strings from my list.

These last sort of three lines-- three, four lines here just do a little bit of arithmetic on it saying, OK, now I have all of the numbers here, what's the minimum out of all of these, and then what's the maximum amount of all these? And then this unique words variable tells me how many unique words do I have in my original tuple.

So this feels sort of generic, so let's run it on some data. So here I have it-- I tested it on some test data. And then I got some actual data. And this actual data that I wanted to analyze was Taylor Swift data. And representing the integer portion of the tuple representing a year and the string portion of the tuple representing the person who she wrote a song about that year.

So some real world data that we're working with here. Very important that we know this information. OK. So with this data, I can run it-- I can plug it into this function that I wrote up here. And I'm going to actually comment this out, so it doesn't get cluttered. And if I run it-- this is the part where I'm calling my function. I'm calling it with this data here. tswift being this tuple of tuples. And what I get back is-- up here, line 38-- is the return from the function being a large tuple. And that large tuple, I'm then assigning it to my own tuple in my program. And then I'm just writing out-- printing out some statement here.

So I'm getting the minimum year, the maximum year, and then the number of people. So I can show you that it works if I replace one of these names with another one that I already have in here. So instead of writing a song about five people, she would have wrote a song about four people. Yay, it worked.

So that's tuples. And remeber or recall-- keep in mind, tuples were immutable. Now we're going to look at a very, very similar data structure to tuples called lists, except that instead of lists being immutable, lists are going to be mutable objects. So much like lists, they're going to contain elements of any type or objects of any type. You denote them with-- you denote a list with square brackets instead of parentheses. And the difference being that they're going to be immutable objects instead of immutable.

So creating an empty list, you just do open close square brackets. You can have a list of elements of different types, even a list of lists. So one of the elements being a list. As usual, you can apply length on a list, and that tells you how many elements are in it. This is going to tell you how many elements are in your list l. So it's not going to look any further than that. So it's going to say, this is an integer, this is a string, this is an integer, this is a list. It's not going to say how many elements are in this list. It's just going to look at the outer-- the shell of

elements.

Indexing and slicing works the same way. So l at position 0 gives you the value 2. You can index into a list, and then do something with the value that you get back. So l at position 2 says-- that's this value there and add one to it. L at position 3, that's going to be this list here. Notice it evaluates to another list.

You're not allowed to index outside of the length of the list. So that's going to give you an error, because we only have four elements. And you can also have expressions for your index. So this-- Python just replaces i with 2 here and says, what's l at position 1? And then grabs that from in there.

OK. So very, very similar to the kinds of operations we've seen on strings and tuples. The one difference, and that's what we're going to focus on for the rest of this class, is that lists are mutable objects. So what does that mean internally? Internally, that means let's say we have a list l, and we assign it-- sorry. Let's say we have a variable l that's going to point to a list with three elements, 2, 1, and 3.

OK. They're all-- each element is an integer. When we were dealing with tuples or with strings, if we re-assign-- if we try to do this line right here, we've had an error. But this is actually allowed with lists. So when you execute that line, Python is going to look at that middle element, and it's going to change its value from a 1 to a 5. And that's just due to the mutability nature of the list.

So notice that this list variable, this variable l, which originally pointed to this list, points to the exact same list. We haven't created a new object in memory. We're just modifying the same object in memory. And you're going to see why this is important as we look at a few side effects that can happen when you have this.

So I've said this a couple of times before, but it'll make your life a lot easier if you try to think of-- when you want to iterate through a list if you try to think about iterating through the elements directly. It's a lot more Pythonic. I've used that word before.

So this is sort of a common pattern that you're going to see where you're iterating over the list elements directly. We've done it over tuples. We've done it over strings. So these are identical codes. They do the exact same thing, except on the left, you're going from-- you're going through 0, 1, 2, 3, and so on. And then you're indexing into each one of these numbers to get

the element value. Whereas on the right, this loop variable i is going to have the element value itself. So this code on the right is a lot cleaner.

OK. So now let's look at some operations that we can do on lists. So there's a lot more operations that we can do on lists, because of their mutability aspect than we can do on tuples or strings, for example. So here's a few of them. And they're going to take advantage of this mutability concept. So we can add elements directly to the end of the list using this funky looking notation L.append. And then the element we want to add to the end. And this operation mutates the list.

So if I have L is equal to 2, 1, 3, and I append the element 5 to the end, then L-- the same L is going to point to the same object, except it's going to have an extra number at the end. 5. But now what's this dot? We haven't really seen this before. And it's going to become apparent what it means in a few lectures from now. But for the moment, you can think of this dot as an operation. It's like applying a function, except that the function that you're applying can only work on certain types of objects.

So in this case, append, for example, is the function we're trying to apply. And we want to apply it to whatever is before the dot, which is the object. And append has only been defined to work with a list object, for example, which is why we're using the dot in this case. We wouldn't be able to use append on an integer, for example, because that sort of function is not defined on the integer.

So for now, you'll sort of have to remember-- which are functions that work with a dot and which are functions like [? ln, ?] that aren't with a dot. But in a couple of lectures, I promise it'll be a lot clearer. So for now, just think of it as whatever is before the dot is the object you're applying a function to, and whatever is after the dot is the function you're applying on the object.

So we can add things to the end of our list. We can also combine lists together using the plus operator. The plus operator does not mutate the list. Instead, it gives you a new list that's the sum of those two lists combined. So in this case, if L1 is 2,1,3 and L2 is 4, 5, 6, when we add those two lists together, that's going to give us an entirely new list leaving L1 and L2 the same. And that's why we have to assign the result of the addition to a new list. Otherwise, the result is lost.

If you want to mutate a list directly and make it longer by the elements within another list, then

you can use this extend function or extent method. And this is going to mutate L1 directly. So if L1 was 2,1,3, if you extend it by the list 0,6, it's just going to tack on 0,6 to L1 directly. So notice L1 has been mutated.

So that's adding things to lists. We can also delete things from lists. We don't just want to keep adding to our lists, because then they become very, very big. So let's see how we can delete some items from our list. There's a few ways. First one being can use this del function. And this says delete from the list L the element at this index. So you give it the index 0, 1, 2, or whatever you want to-- whatever index you want to delete the element at.

If you just want to delete the element at the end of the list, that's the farthest right, you do L.pop. If you want to remove a specific element-- so you know there's somewhere in your list there's the number 5, and you want to delete it from the list-- then you say L.remove and you say what element you want to remove. And that only removes the very first occurrence of it. So if there's two fives in your list, then it's only going to remove the very first one.

So let's take a look at this sort of sequence of commands. So we have first L is equl to this long list here. And I want to mention that all of these operations are going to mutate our list, which is why I wrote this comment here that says assume that you're doing these in order. So as you're doing these in order, you're going to be mutating your list. And if you're mutating your list, you have to remember that you're working with this new mutated list.

So the first thing we're doing is we're removing 2 from our list. So when you remove 2, this says look for an element with the value 2 and take it away from the list. So that's the very first one here. So the list we're left with is just everything after it. Then I want to remove 3 from the list and notice there's two of them. There's this 3 here and there's this 3 here. So we're going to remove only the first one, which is this one here. So the list we're left with is 1,6,3,7,0.

Then we're going to delete from the list L the element at position 1. So starting counting from 0, the element at position 1 is this one here. So we've removed that, and we're left with 1, 3, 7, 0. And then when we do L.pop, that's going to delete the element furthest to the right. So at the end of the list, which is that 0. So then we're left with only 1, 3, and 7. And L.pop is often useful, because it tells you the return value from L.pop is going to be the value that it removed. So in this case, it's going to return 0.

I want to mention, though, that some of the-- so these functions all mutate the list. You have to be careful with return values. So these are all-- you can think of all of these as functions that

operate on the list. Except that what these functions do is they take in the list, and they modify it. But as functions, they obviously return something back to whoever called them. And oftentimes, they're going to return the value none.

So for example, if you are going to do L.remove 2, and you print that out, that might print out none for you. So you can't just assign the value of this to a variable and expect it to be the mutated list. The list got mutated. The list that got mutated is the list that was passed into to here. We're going look at one example in a few slides that's going to show this.

OK. Another thing that we can do, and this is often useful when you're working with data, is to convert lists to strings and then strings to lists. Sometimes it might be useful to work with strings as opposed to a list and vice versa. So this first line here, list s takes in a string and casts it to a list. So much like when we cast a float to an integer, for example. You're just casting a string to a list here. And when you do that up this line-- so if this is your s here-- when you do list s, this is going to give you a list-- looks like this-- where every single character in s is going to be its own element. So that means every character is going to be a string, and it's going to be separated by a comma, so including spaces.

Sometimes you don't want each character in the list to be its own element. Sometimes you want, for example, if you're given a sentence, you might want to have everything in between spaces being its own element. So that will give you every word in the sentence, for example.

In that case, you're going to use split. In this case, I've split over the less than sign. But again, if you're doing the sentence example, you might want to split on the space. So this is going to take everything in between the sign that you're interested in-- in this case, the less than sign-- and it's going to set it as a separate element in the list. So that's how you convert strings to lists.

And sometimes you're given a list, and you might want to convert it to a string. So that's where this join method or function is useful. So this is an empty string. So it's just open close quote right away. No space. So this just joins every one of the elements in the list together. So it'll return the string abc. And then you can join on any character that you would want. So in this case, you can join on the underscore. So it'll put whatever characters in here in between every one of the elements in your list. So pretty useful functions.

OK. Couple other operations we can do on lists-- and these are also pretty useful-- is to sort

lists and to reverse lists and many, many others in the Python documentation. So sort and sorted both sort lists, but one of them mutates the list and the other one does not. And sometimes it's useful to use one, and sometimes it's useful to use the other.

So if I have this list L is equal to 9,6,0,3, sorted-- you can think of it as giving me the sorted version of L-- gives you back the sorted version of L. So it returns a new list that's the sorted version of the input list and does not mutate L. So it keeps L the exact same way.

So this will be replaced by the sorted version of the list, which you can assign to a variable, and then do whatever you want with it. Like L2 is equal to sorted L, for example. And it keeps L the same.

On the other hand, if you just want to mutate L, and you don't care about getting another copy that's sorted, you just do L.sort. And that's going to automatically sort L for you, and L now-- L is now the sorted version of L. Similarly, reverse is going to take L and reverse all the character-- all the elements in it. So the last one is the first one, the second to last one is the second one, and so on.

So lists are mutable. We've said that so many times this lecture. But what exactly does that mean? What implications does that have? Once again, this next part of the lecture, Python tutor. Just paste all the code in and go step by step to see exactly what's happening.

So lists are mutable. As you have variable names-- so for example, L is equal to some list-- that L is going to be pointing to the list in memory. And since it's a mutable object, this list, you can have more than one variable that points to the exact same object in memory. And if you have more than one variable that points to the same object in memory, if that object in memory is changed, then when you access it through any one of these variables, they're all going to give you the changed object value.

So the key phrase to keep in mind when you're dealing with lists is what side effects could happen? If you're mutating a list, if you're doing operations on lists, what side effects-- what variables might be affected by this change? Let's come back down to earth for a second. This will wake a lot of people up.

So let's do an analogy with people. Let's say we have a person. A person-- this case, Justin Bieber-- is going to be an object. I'm an object. I'm like the number three. Bieber's an object. He's like number five. Different objects. Were both of type people.

OK. Let's say a person has different attributes. Let's say we can-- let's say he gets two attributes to begin with. He's a singer and he's rich. I can refer to this person object by many different names. His full name, his stage name, all of the fan girls call him by these names, people who dislike him call him by other names that they didn't put up here. But he's known by all these different names. They're all aliases or nicknames that point to this same person object.

OK. So originally, let's say I say Justin Bieber is a singer and rich. Those are the two attributes I've originally assigned to him. And then let's say I want to assign a different attribute to him and say Justin Bieber's a singer, rich, and a troublemaker. I'm being kind here.

OK. So if I say Justin Bieber has these three attributes-- so it's the same person I'm referring to-- then all of his nicknames are going to refer to this exact same person. So all of his nicknames or aliases will refer to the same person object with these changed attributes. Does that makes sense? OK.

So that sort of idea arises in lists. So a list is like a person object whose value-- whose attributes can change, for example. And as they change, all of the different aliases for this object will point to this changed object.

So let's see a few examples. I apologize if this is a little small, but this I basically copied and pasted from the Python tutor, which is just from the code from today's lecture. So I have these lines of code here. The first couple of lines really just show what happens when you're dealing with non-mutable objects. So with non-mutable objects, you have two separate objects that get their own values, and that's it. End of story.

With lists, however, there's something different that happens. So I have warm is a variable. And it's going to be equal to this list. So warm is going to point to this list here. Red, yellow, orange. It contains three elements.

Hot is equal to warm. It means I'm creating an alias for this list. And the alias is going to be with this variable hot. So notice warm and hot point to the exact same object. So on line 8 when I append this string pink to my object, since both of these two variables point to the exact same object, if I'm trying to access this object through either variable, they're both going to print out the same thing. And that's the side effect. That's the side effect of lists mutable.

If you want to create an entirely new copy of the list, then you can clone it, which sounds really

cool. But really, it's just making a copy of the list. And you clone it using this little notation here, which is open close square brackets with a colon. And we've sort of seen this notation here. And this tells Python this is 0-- sorry. This is 0 and this is length. Cool.

But it basically says take every element, create a new list with those exact same elements, and assign it to the variable chill. So here, if I originally have cool is equal to blue, green, gray right here, when I clone it on line 2 with that funky notation, I'm creating a new copy of it. And then on the next line when I'm appending another element to the copy, notice I'm just altering the copy. The original stayed the same, because I've cloned it. So if you don't want to have the side effects-- side effect issue, then you should clone your variable-- your list.

So let's see a slightly more complicated example where you're going to see the difference between sort and sorted in the context of this mutability and side effects issue. OK. So once again, let's create this warm is equal to red, yellow, orange. So that's what warm is going to point to, this list. And then sorted warm is equal to warm.sort. So .sort mutates. So as soon as I do that, that list warm is now the sorted version of it. And notice that I've assigned the return of this to sorted warm. And the return is none, because L.sort or .sort mutated the list. It didn't return a sorted version of the list. It mutated the list itself.

OK. So when I print warm and I print sorted warm, I'm printing the mutated version and then this one here. Sorted, on the other hand, returns-- it doesn't-- sorted does not sort the list that's given to it. And instead, it returns a sorted version of the list.

So in this case, if cool is equal to these three colors-- gray, green, blue-- if I do sorted cool, it's going to return the sorted version of that list, which is blue, green, gray. And it's assigned to the variable sorted cool. So when I print them, it's going to show me the two separate lists. One being the original unsorted one, and one being the sorted version.

Last ones a little bit more complicated, but it shows that even though you have nested-- even though you can have nested lists, you still-- you're not-- you don't escape this idea of side effects. So first, I'm going to create warm is equal to these two colors, yellow, orange. So warm points to these two colors. Hot is equal to this one list-- a list with one element. Bright colors is going to be a list. And the element inside the list is a list itself.

So since it's a list-- this is your list, and the element inside here, which is a list itself, is actually just pointing to whatever warm is. That object. Then I do-- then I append hot to my bright colors. So the next element here is going to be another list, which means it's just pointing to

this other list here. It's not creating a copy of it.

So each one of these elements here is actually just pointing to these two lists here. So if I modified either one of these, then bright colors would also be modified. So let's say I add pink here to my hot list. We have red and pink. Then notice that bright colors-- the first element points to this list, and the second element points to this list, which I've just modified.

Last thing is-- I'll let you try this as an exercise in Python Tutor-- but the idea here being you should be careful as you're writing a for loop that iterates over a list that you're modifying inside the list. In this case, I'm trying to go through the list L1. And if I find an item that's in L1 and L2, I want to delete it from L1.

So 1 and 2 are also in L2. So I want to delete them from L1 and be left with 3, 4. However, the code on the left here doesn't actually do what I think it's doing, because here I'm modifying a list as I'm iterating over it. And behind the scenes, Python keeps this-- keeps track of the index and doesn't update the index as you're changing the list. So it figures out the length of the list to begin with and how many indices it has. It doesn't update it as you're removing items from the list.

So the solution to that is to make a copy of the list first, iterate over the copy, which will remain intact, and modify the list that you want to modify inside the loop. So please run both of these in the Python Tutor, and you'll see that what ends up happening is on the left, you're going to skip over one element. So your code-- so that's going to be the wrong code. All right.