PROFESSOR: So let's start. I mean for the first question you can answer what is tuple. You can go to that later. Can a tuple contain a list? What do you think? Can a tuple contain a list?

AUDIENCE: Yes.

PROFESSOR: Yes, it can. So we'll go through the examples later. And let's see this first example. We have two tuples. The tuple B contains tuple A, as well as a list. So it can't populate. Let's bring those lists, too.

So for the first example we try to access the elements 0 and 2 of the tuple A. And let me use minus 1 to access which element?

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK. And 0 is the first element, of course. And remember now we are using 0, 1. OK, what is tuple B? It's a two-dimensional tuple.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, exactly. It has a list and a tuple inside. So the first element is a tuple, and the last element is the list. So now it's sort of like two-dimensional. But if you look at the first element of it, the MIT word, that doesn't have any dimension. Does it? It's a string. So you can again access, but not this way. But still you can access.

And we are using this notation to access a portion of the list, a part of the tuple, actually, here. It's called slicing. So here 0 to 1 gives you which element? Just one element right? It's the first element. 0 to 2 gives you 1 and 2 OK? This is an interesting usage of the slicing. This is used to access the whole tuple. Now you

might wonder why we need them.

We could just-- the handouts are here. So why we need to copy or why we need to access the list through the column operator like this? We'll look at that later. It's very essential when we have lists. For the timing it's OK when tuples are immutable. So it doesn't have a special usage. But later on we'll be using it.

Then if you look at this part, this is the most interesting part. How do iterate through a tuple? You use a FOR loop and you call like item in tuple, the element in tuple. The element is initialized with every element when it goes through iteration.

At every iteration it will be instantiated with the corresponding element in the tuple. But you're using the same name anyway. It's item. This is equivalent to accessing the tuple like this. Going one-by-one in the range and accessing their corresponding element in the tuple. So this is the simpler way to access that. Do you have any questions in that part? Yeah?

AUDIENCE: For that part, don't, usually if you have a range you have to [INAUDIBLE] name because is not--

PROFESSOR: OK, that's correct. . OK. Suppose a tuple has 3 elements, or for example in this case tuple A, how many elements are in the tuple A? 3 elements. So what would be the length of tuple A?

AUDIENCE: 3.

PROFESSOR: 3. But what would be the range? 0, 1, 2, right? So you have to type 0 to 3. So that's why I put that, OK? That is 3, OK?

PROFESSOR: You.

AUDIENCE: So you just say a range like the tuple is, it seems like the first one--

PROFESSOR: First the 0 if you don't specify it the same way.

OK. Now let's go through the list. So if you go through the questions, what is the

difference between a list and a tuple? What is the difference between a list and a tuple? Anyone? Yeah?

AUDIENCE: Tuples are immutable.

PROFESSOR: OK. Lists are mutable. And because it's mutable it takes special functions to access the elements, add elements, and modify the list itself. How would you add an element to a list? Suppose you have list A. How would you add 3 to this list?

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK. Let's start our print. OK. That's good. OK? How would you remove the element from the top of the list? Or the last element, how would you remove this 3 from the list?

AUDIENCE: [INAUDIBLE]

PROFESSOR: This one? Let's see. You know how to check the specification of a function, right? Sorry. So it's not the right function, right? Because it says it removes the first occurrence of a particular value you are parsing to that function. OK, there's another function. It's called list.pop. That removes the last inserted value, so it pops from the top. So remove can be used to remove a particular element. So here suppose we want to remove 1. So we could say list A, remove 1.

OK. So in this example, actually, I have commented out this line. This will be an error if I execute. Why? Why will it give you an error?

AUDIENCE: [INAUDIBLE]

PROFESSOR: It doesn't exit. Great. So the problem here is we can't access an element in the list unless we explicitly assign something or create something.

But suppose we want to pass a list of 10 elements to a program so that it can assign values on the fly. And it can modify. OK? So how would you pass an empty list of 10 elements? OK. The problem is if you're passing something like this, you need to know how this list should be instantiated. So you should know what are the initial

values of these elements.

Suppose you want to create a list of ten 0's. Then the easiest way to create the list is-- So you would create a list with ten elements. But If you don't want to waste memory, or if you don't want to keep anything particular, you could do something like this.

Now this will create a list of empty list. And there'll be ten such empty lists. OK? And you could pass this list to a program so that it will assign the elements. OK?

You can iterate to the list the same way. But lists are interesting. Why? They are mutable. So they can be actually used to create interesting mathematical objects. For example, like matrix.

So how would you create a matrix using lists? Any ideas? How would you create, say a 2 by 2 matrix? If you consider this example, this creates a matrix 4 by 2. So that would be like 4 rows and 2 columns. But how can you go and access the inner element or inner matrix? So if you suppose you want access 2, 3, that is the third row, then how would you access? How would you call M?

AUDIENCE: M3?

PROFESSOR: M3. M3 gives you 2, 3. Suppose you want to access the 3 and the 2, 3. Then we--

[INTERPOSING VOICES]

PROFESSOR: Sorry.

AUDIENCE: Why does a 3 give you [INAUDIBLE]?

PROFESSOR: Oh, sorry. My bad. Yeah. OK. This is computer science, not maths. You got it right, why it's not 3, it's 2?

AUDIENCE: Yeah. I thought you were talking about 3 and 4.

PROFESSOR: OK. That's fine. So I'm asking now how would you access the 3 in the 2, 3 box?

AUDIENCE: So I need to do another parentheses?

PROFESSOR: Not a bracket.

AUDIENCE: OK.

PROFESSOR: OK. That is all. So you can access matrices like this. OK. Great. We are on time. So this actually gives you a summary of functions. Associated functions. For example the pop I explained earlier. Remove removes a particular element.

Extent, that's interesting. Can you tell me what it does here, the method extent on list? What does that method take? It takes another list. So which means it's going to merge the list you are passing with the list you're calling the method on, that is [UNINTELLIGIBLE]. So it's going to combine those lists. Let's see. So this corresponds to this. So two lists are combined. So this method is actually useful, right, because you don't have to write yourself. Most of things you can easily do.

Now the question is, why tuple didn't have these methods? Why? Why we didn't have these methods for tuples?

AUDIENCE: [INAUDIBLE]

PROFESSOR: They're immutable. So you can't extent, you can't [UNINTELLIGIBLE]. So even if you extent, what would happen? It's going to create a new tuple. Actually, you can do that. But like this. For example, you have tuple A which is 1, 2. And suppose you have tuple B -- 3, 4. If you want to combine them, how would you do it? Tuple A plus tuple B. OK? That concatenates. This operator is all loaded for tuples. OK? So lists are mutable so you can do all of these things.

Let's look at a few other examples, too. OK. There's an interesting part here. Lists are mutable. That we understand. But how these lists are actually stored in the memory, For example, suppose you have a list lst, sorry, 0, 1, apple. How do you think it's stored in the memory? So you have 3 elements. But these elements are actually not stored in the list itself. They actually in the memory. 0, 1, apple. But you have the pointers to those lists. Sorry, to those elements. So actually you can

modify these values without changing the list itself, because the list actually points to this part, the container.

So in this example you have list A and B. But the list B actually contains list A. OK? Since list A is mutable you can go and change it. So here first you print list A and B, but you're going to change list A's 0th element. After changing the element the list A becomes first. Its first element changed to 88. OK. Fine.

What do you think the list B would be? OK, let's print. It contains the 88, right, the modified value. Why? Because list B actually had a pointer to list A. Which again had pointers to other elements. And we modified this value to 88. So you're still accessing list B. OK? The pointers are not changed.

That we can see here, in the second example, the problem of aliasing. So we have list X. And we call it by list Y as well. And we are now changing the list Y. Sorry. We are changing the list X. But it's going to affect list X as well as list Y. See? Because it's just a reference. The actual object is this. You can call it by two different names. It doesn't matter. But the actual object is here. That's where we need this operator.

So if you just point, OK, for example we had list X, if you just assign list Y to list X, it's not going to copy the elements, just going to copy the reference. Which means list X and list Y, both would be pointing to the same object. If you want to copy these elements and put in a new one, say, list Z, then we need to copy one-by-one every element. For that we can use this operator. It's called full slicing. It's not going to modify elements. It's going to copy the whole list of elements.

And in the same line, just another small example that tells you how to cast a list into tuple, and a tuple into list. So you had a tuple, one for apple [UNINTELLIGIBLE] and you cast into list. Now the type is list. It's a very simple operation. Can you do that yourself? How would you do it if you want to write a function to cast, how would you do that? You go through element by element and add it to a tuple. And then concatentae two tuples. Right? So I guess you can write probably as a homework, try that. Try how to convert a list to a tuple without explicitly calling the casting function. OK.

Similarly you can convert a tuple back into list. So it's pretty straightforward. And suppose you're getting an input from the user as a tuple. then suppose you want to make it a list. You don't have to write your function. You can just call this casting operator. So it'll be-- it will save some time. OK. Great.

Let's go to our dictionaries now. So we have a third data type, dictionaries. Oh, you didn't get the second one? So we have another data type, dictionaries. And why do we need dictionaries in the first place? Do we need another data type? Can we just get away with tuple and list? Yes. Because you can actually make a dictionary out of lists. And Professor Grimson went through that in the lecture. If you want, you can come and ask in the office hours. But first try whether you can do it yourself.

But the problem is, actually not just the dictionaries, but the high-level data structures are actually available in Python as building data structures or as classes. Just because they will have these methods on these data structures implemented. So it would say some time for us. Plus those methods are guaranteed to be state of art. For example, if you want to go through all the lists, then the search function or the lookup function that you write may not be that efficient, right? But they might use those efficient algorithms in the standard implementation.

So it's why you should always look for the standard data types. If you can't you can write yourself one. But it's good to use the available data types so that they would have this efficient method implemented.

OK. So in this dictionary, we have two elements. One is key, one is value. OK? So these keys actually have a special property. What is that? The keys should be an immutable object. So you can actually have a tuple as a key. You can have a string as a key, but not a list as a key. So you need an immutable object for the keys.

What about the values? They can be anything. Values can be even a dictionary. It can be a list. But the keys must be mutable. Do these keys need to be unique? Yes. No? OK. The problem is this. Dictionaries have the same type, I mean structure. This is a key. And this key points to a place in the memory. Suppose you call it key

A. OK? If you assign another value to key A, that will actually replace the content in the memory, right? So that's why keys are going to be unique, It's by construction, by the construction of the dictionary the keys are going to be unique. Because if you want to assign something different, you have to call it by a different key, call it by a different name. Say key B.

What about the values. Do they need to be unique? No, of course. Otherwise no use of key, and no use of dictionaries, right? What is the order of the items stored in the dictionary? Can you give a guarantee like in lists? In lists items are stored from zero to the left, right? Length minus 1, actually. What about dictionaries? Surely can't guarantee the order, but you can modify the order. OK. So the thing is first, if you look at this example, first we have the staff dictionary. And when I pin the length it gives me 3, fine. Then I'm doing three things.

First I change the address by calling its key. And dictionaries are mutable so you can change their values. Then also we are adding a new element. Fine. If you wanted a new element you call it by the key that you already assigned. If the key already exists you will just modify the value. Otherwise it will create a new key and add that value.

You can also check whether this particular element is in the dictionary. But this must be the key. Whatever you are calling here should be the key, not the value. It won't search for the value. It will search only for the key, [UNINTELLIGIBLE] for that particular key because you access the values through keys.

Again here if it is not in the list, actually I'm adding an element-- OK. OK. To compare in the list, suppose I have a list 0, 1, 2. If I want to check whether 1 is in the list, how would I check? Do you have to go through element by element? No, there's a shortcut. If 1 in list, right? If I want to check whether 1 is not in the list, if I want to negate this, I would write it 1 not in list. We just ignore the words. It's pretty much like English. OK? We just remove the words. OK.

In the dictionary we do the same thing through keys. If key A and, OK I'll call it, say, D1. OK? And you actually call or you'd search for the key, not the other way.

OK. There is an interesting part here. If you want to modify the order-- actually I tell you that the order C earlier, we couldn't guarantee the order, right? It was not the order we typed because it starts at [UNINTELLIGIBLE] because we added at the end actually, right? So actually you can't guarantee the order.

But we can sort it. But how do you sort it? We call the keys and we sort the keys. Because remember, every values-- if you want to access a particular value, you access it through key. Actually, if you call the dictionary, it doesn't know where these values are. Just you have to go to the key to access the value.

So if you want to do something in the dictionary itself, you can do that only on keys. So if you want to sort the dictionary, you sort it by keys. So you call the keys method for the dictionary-- so it returns the list of keys and you sort them. This is called chaining methods. So I have chained the methods. I would have like two methods, right? I call the keys. This first part actually returns to this top key. Then you sort them. Yes?

AUDIENCE: It doesn't look like they're in alphabetical order?

PROFESSOR: Oh, on this? Sorry. Let's see. OK. Let's see. Ooops. OK. Now they're sorted. OK. What was the problem? What was the problem there? Why it wasn't sorted?

AUDIENCE: You have to do it in two lines.

PROFESSOR: Sorry?

AUDIENCE: You have to call up the keys and then sort it in a separate line?

PROFESSOR: OK. The problem is in the logic. Actually, when you call this function, method, you're actually sorting the keys returned by this method. You're not actually going to sort of the dictionary itself. You're sorting only the list that was returned, right? The list of keys. Although the dictionary is mutable, it wasn't sorted. Do you see the problem? So that's why when they called keys equal to staff.keys I'm getting a list of keys. Then I'm sorting that list. And I'm printing that list.

But if you want to go in a particular order, if you want to access the dictionary in a particular order, what could you do is you could do something like keys is equal to staff.keys. Then you can store keys. Then for k in keys you can go and iterate now, right? So you can say print k and staff k. OK? So you can do this.

But actually, Python provides a way to iterate what keys and value pairs. That you do by calling both elements? All the method items. The items return a list of key-value pairs. If you call D1.items you get a list of key-value pairs. You see that? And we are going to iterate through individual elements. So first we start with this and then second this. So that's a simpler way to access every elements in the dictionary. Great. OK.

Now we have an interesting part, recursion. What's the principle behind recursion? Anyone? What's the idea of recursion? Yes?

AUDIENCE: The cause itself is [INAUDIBLE] base case. And it saves a lot of money.

PROFESSOR: Yes. The idea of recursion is, if you have a problem, try to express the problem in a simpler version of the same problem. So if you want to find factorial n, try to express it in terms of factorial n minus 1. So you could keep on doing this till you come to factorial 1 for which you know the explicit answer. Right? So you try to express the problem in its simpler form.

It would be useful in many cases. Actually, in your next piece you do have the problem. But be mindful. What's -- what makes it possible for you to use the recursion? Only if you can express in terms of the simpler version. Otherwise you can't. This is probably quite like the induction you might have studied, mathematical induction. I don't know whether you studied it in high school, but it's quite like that.

OK. So there are two parts in recursion. The first one is the base case. So for a factorial problem we can express the factorial, say nth factorial, as any n in to n minus 1 factorial. OK? So this would be our recursive case. So what is a recursive case? Suppose we want to define a function factorial A. OK? So what will be our recursive case?

AUDIENCE: A is greater than 0?

PROFESSOR: I'm asking the recursive function. What would be that?

AUDIENCE: [INAUDIBLE]

PROFESSOR: n into 2. Factorial n minus 1 right? And you just return this. What would be your base case? If n is equal to 0, we know factorial 0 is 1, right? What is factorial of 1? 1. OK? But why in this program we didn't have that particular line. Why we didn't have n equals 1? Why we didn't have that?

AUDIENCE: Because it always goes to 0?

PROFESSOR: It always goes 0, right? Because you can express 1 in terms of 0 as well, right? So you don't need to actually write this explicitly. Why? Because your recursive function only depends on its previous value. So you need only one value in advance Which means you need only one value for your base case. So you can understand this program, right? I'm not going to go through that. But anyway when you write a program always check its base case. So you have to start with factorial 0 for this case. OK? And check one by one. Then you would know whether the program is running correctly or not.

So always start with a simpler case. But remember in Fibonacci series. OK. for Fibonacci series can you give me the recursive function.? Yes. What is Fibonacci series?

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK. Yeah. That's great. So return say F of n minus 1 plus F of n minus 2. What is the Fibonacci series? You start from 0, you add these elements. So 1. You add last two elements, 2. You add last two elements, 3. You add last two elements, 5. So you add last two elements.

So if you want to find F of n, you return F of n minus 1 and plus F of n minus 2. But here since you have two elements, or you need to access two previous elements,

you need to define you base case accordingly. So for your base case, F of 0, is what? Actually, that depends on whether you start here or here. OK? You could do this.

I'll go through an interesting recursive example. OK. It's called a recursive exponentiation. So I actually you can do an exponentiation through recursive multiplication. Suppose you want to find 3 to the power, say, of 10. Then how would you do that? You start by expressing it in terms of its simpler version, right? So it will be 3 into 3 to the power 9. Sorry, 3 to the power 9. OK? So if you want to find that nth power of number M, you would say M into M to the power n minus 1. So now you have your recursive case. What is the base case?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yes, if n is equal to 0. In Python you would test this by two equal signs. If n is equal to 0, then what?

AUDIENCE: Return 1.

PROFESSOR: Return 1. OK? Why we have only one base case here?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Sorry?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yes. And because you're accessing only one previous series, right? Fibonacci series. Similarly, if you want to multiply something-- OK, here what we did is we replaced this operator by recursively using this operator. OK? Can you do the multiplication by recursive addition? So can you replace this operator by this operator? Yes. So how would you do this, for example say 3 into 5? So how would you do it here? Tell me. What would this?

AUDIENCE: 3 plus 3 times 4.

PROFESSOR: 3 plus?

AUDIENCE: 3 times 4.

PROFESSOR: 3 times 4. Here? Come on, it's simple, right?

AUDIENCE: [INAUDIBLE]

PROFESSOR: I'm insulting you. OK. What's the base case? That is interesting. What is the base case if n is equal to 0?

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK. If n is equal to 0, that is 0. But that's fine. But there's another problem. The problem is you don't know whether these values are positive or negative. So you can't just keep on adding like this. This will work if it's positive.

But if it's negative, actually you have to check. Which is sort of tricky if you want to use the same recursive function. Otherwise you could have an if condition here. If say n is greater than 0, do this. Else-if you multiply n by minus 1 you negate that so it becomes positive. And do it the same way and then that's the answer. So you could do that. But if you want to do it in the same recursive case, then you had to follow this. If you have any questions, come to the office hours on that.

The final question is, final example is, the Hanoi example. OK. This is very interesting. So here we have three towers; source, target, and buffer. Suppose you have three disks. You want to move them from source to target. And you can always have a smaller disk on top of a bigger disk. Then how would you move? OK? So can you express this problem in a simpler version?

So now we have three disks. You want to move them to target. So can you express it in a simpler version? Come on. Just in plain English, how would you do it? Without looking at the program. Because Python is very robust. So if you read the program you get it in English. So any ideas? OK. We have to here move the last disk to target, right? But that's the hardest part. So before doing that you have to move this somewhere else. So let's move it to buffer. Done. How would you move this? Now

you can move it to target. OK. Now what would you do?

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK. Sorry. Yeah, OK? No. What would you do?

AUDIENCE: [INAUDIBLE]

PROFESSOR: You can only move this to the buffer, right?

AUDIENCE: [INAUDIBLE]

[? PROFESSOR: Not the target. OK. So we made some mistake earlier. Remember? The problem is because we didn't think recursively. OK. Let's think recursively. Source, target, buffer. Sorry. Suppose we have only one disk. Then you can simply move to target. OK? So a condition is if there's only one disk, just move it to the target. So if n is equal to 1 we always move from move source to target. OK?

Suppose you have two. Then what would you do? You move the top one to buffer. OK? So move source to buffer. Then you move the next disk from source to target. So now it's source to target. And finally you move it from the buffer. All right? Now we have a very nice recursive case. If you have two, we move from source to buffer, then source to target, then from buffer to target.

So if you look at this example I'm checking if n is equal to 1. If it is 1, just I'm moving from source to target. If it is greater than n, what would I do? And that first assert statement is just there to make sure that n is greater than 0. Otherwise it's meaningless, right? You need to have this. You can't have negative this.

So what would be the next step if you have more than one disk? Suppose you have two disks. Then what would you do? You move the top disk or whatever on the top to the buffer. For this simple example it was just one disk.

But suppose you had more disks on top of the last disk? So you would have moved all of them to buffer. You don't have to worry how you move it. But you have to move it anyway. After moving that, you can leisurely move this big disk to the top, to

the target. OK?

Finally, you bring back all of them here, right? For you first operation, for your first operation, from source to target, you have to move all of them, the [UNINTELLIGIBLE]. You could have used target as your buffer. OK? So that's why in this line I'm using source as source. But the second argument, which should have been the target, is now buffer because my current target is buffer for the top n minus 1 disk. But my buffer is now the target. OK. Because I can use the target as buffer for the movement. Then I moved from source to the target, the last big disk. OK? For that I can use buffer as my buffer. Then finally I bring back from buffer to target using source as my buffer. OK?

So the thing is if you want to have a recursive problem you can do it in two ways. One thing, you can start thinking how to express this problem in its simpler version like how you thought about the Fibonacci series, for instance. Or else, you can start from its base case, the most fundamental situation. So here it's n equal to 1, but this doesn't give you enough context. So you go to the next level, n is equal to 2, and you have the answer. Right?