[SQUEAKING]

[RUSTLING]

[CLICKING]

ANA BELL: So let's start today's lecture. Today, we're going to be talking about the idea of iteration. And iteration is another way we're going to add control flow to our programs. But before we do that, let's do a little bit of a recap on-- sorry, let's do a little bit of a recap of what we've done so far last lecture because last lecture, we actually introduced a different mechanism for control flow, branching.

And the control flow is basically a way for us to tell Python not to go systematically through the code. Branching was a way for us to tell Python, hey, based on some condition being true or false, either evaluate some set of code or another set of code, which was not going linearly. We were actually kind of skipping around through the code.

So that's what we learned at the end of the lecture. But we also learned about input and output-- so a way for us to write interactive programs. And we learned about a new data type, the string. So the string was a sequence of characters. Hopefully, you got a chance to do a little bit of exercises on MITx as practice for today's quizzes with strings and branching.

OK. So in branching, what did we learn? We talked about how to actually add a branching point in our program. So we did that using these particular keywords. So when you type them in your program in the file editor, you'll see that they turn a different color. That tells you it's a special word in Python.

And these keywords are how we told Python to put a branching point. And the colon of ended the branching conditional. And then anything that was indented as part of that conditional was code that would be executed when that condition was true.

So I'm just going to quickly go over these, each one of these boxes. So the first one up here was the simplest way that we could add a conditional to our program. It basically said, hey, go through the program, when you reach this if condition, Python would check the condition and say, if that condition is true, execute the code that's indented as part of that block. If the condition was not true, don't do anything, just carry on with the remaining program.

If we wanted to do something else-- so if the condition was not true, if we wanted to do something else, we added this else clause here. And the else also has some sort of code indented as part of its code block. And that code would be executed when that condition was false.

OK. So that was a really simple if or if/else code structure. But sometimes, we want to have code that checks for many conditions, not just one. That's where the elif structure came in. So we would have an if condition that starts our code block. If that condition was true, as usual, we execute the code that's part of that block. Else if, so elif, we could insert another condition. And Python would say, OK, well if that one wasn't true, let me check if this next one is true. And then we would execute the code that's part of that code block.

We can chain as many of these elifs as we want together. And Python will evaluate the very first one that it finds true that's part of this chain, even if more than one is true. It is possible none of those conditions were true, in which case Python would basically skip over all of them and do nothing, enter none of those code blocks.

If you wanted to have a structure where if none of those conditions were true you wanted to do something, you could put an else at the end of a whole chain of if, elif, elif, elif's. And the else would be executed when none of those conditions are true. So hopefully, this is just recap.

One tricky thing to remember is the if starts a code block. So the if can have an else associated with it or it can have an elif, elif, elif, and an else associated with it. But if condition and then followed by another if condition, both of those code blocks could potentially be executed because the if's are independent. It's not an else situation. They're just another if code block that gets started.

Again, just to reiterate, the way you told Python which code to execute when the condition is true is by indentation. And indentation is something you have to do. It's not optional in Python.

OK. So let's take what we've learned so far and code up a really simple game. So this is a very simple variation of the Lost Woods in *Zelda,* my version of it. Let's say it's a trick level where you have your character and they enter the Lost Woods. They're presented with this screen.

And the trick here is you ask the user if they want to go left or right. If they say right, you're basically going to present them with the exact same screen all over again. So it's kind of representing that they're lost in the woods. And as long as they say I want to keep going right, I want to keep going right, I want to keep going right, they're basically going to see the same screen over and over again. And the trick to getting out of the woods is to say I want to go left. So no matter how far, how many times they've said right in a row, as long as they type in left, they're out of the woods.

So let's try to code that up with what we know, just conditionals. We have an if/else. The if says, if the user exits right, we're going to do something. And otherwise, we're going to say that the user said left, or something else, or exit. And then we're going to tell them that they've exited successfully.

All right. Now, if they said exit right, what do we do? Well, we're going to show them the exact same thing again. So we're setting the background to the same woods background. And then they're presented with the choice all over again. Do you want to exit right? Or do you want to exit left? So if they say exit right, we would do something. And otherwise, we would tell them they successfully exited.

Well, what if they exited right? If they exited right, then we would do something again, basically present them with the same situation. So we would set the woods back around again. And we would ask them if they want to go right or left again. And otherwise, if they said left, they exit.

So we already see a problem, right? How deep do we make this nested loop situation? Here, we already have three, in the case the user said, I want to go right three times in a row. But we don't know how persistent the user will be. So how do we know, when we're writing our code, how deep to make this nested loop? We don't. We won't be able to really code this up very well with what we know so far.

And so that's the motivation for introducing iteration because the situation on the previous slide fits really well with some task we want to repeat multiple times, as long as some condition is true. In our case, the condition is the user says, I want to exit right. So while the user keeps saying exit right, show them the woods background and ask them again which way do you want to go.

And so while that's true, just repeat this set of things-- check that they said exit right, show them the woods background, ask them again, check that they said exit right, show them the background, ask them again. And if at any point they'd say I don't want to exit right, we break out of this loop and we rejoin the rest of the program. That's the terminology we use with if statements. We set the background to the exit background and they're out of the woods.

So this sets the scene for a while loops. Here's another example of while loops in the context of watching a show. So if we want to start a new show on Netflix and we want to watch all episodes of the show in one shot, we're going to tell Netflix we're starting a new show.

And while there are more episodes to watch in this show, we're going to keep watching the next episode. So if there are no more episodes to watch, then we're done. Python-- not Python, Netflix will suggest three more shows like this one. And while there are more episodes to watch-- so yes, there are more episodes to watch-- we're going to play the next episode in sequence. So that's the idea that we're trying to get at with while loops.

In Python, this is how we code them. So we start a while loop with the keyword while. So this, again, will turn blue in Python because it's a special word. Some condition is true. So this is, again, some expression or something that will evaluate to a Boolean, like we talked about in last lecture. Colon-- and colon tells Python we're starting a code block that's part of the while loop being true. And as usual, the code block means we're going to indent these lines of code. So whatever we want to execute when the condition is true will be indented.

When the indented statements are finished executing, Python automatically goes back and re-checks the condition. So it re-checks whether the condition is true or not. And this is done behind the scenes. When you code up a while loop, when you type in the keyword while, Python will automatically do this behavior.

It will check the condition. It will execute the lines of code indented. And then it will go back and check the condition again. If it's still true, it will execute the lines of code indented again. And then it will check the condition again. So it's not something you have to code up. You don't have to tell it to go back.

As long as you're writing this while loop, Python will automatically do that sequence of steps for you. So when the condition becomes false, Python will no longer execute the stuff inside, the stuff that's indented inside the while loop. And it'll go rejoin the rest of the program at the same indentation level as the while loop.

So notice that the condition is kind of something that's dependent on-- or sorry, the fact that we're doing this code over and over again depends on this condition being true. So if the code inside is not ever changing anything related to our condition, then it's very likely-- it's actually for sure that this loop will execute infinite number of times. So this is the pitfall of while loops. It's possible that, if you're not careful, your code will execute an infinite number of times. And it'll just never terminate. And I'll show you how to deal with that in a couple of slides.

So let's try to code up this Lost Woods program just with a while loop. So here, we've got our question that we ask the user. Do you want to go left or right? And we're going to grab the user input as a string and save it in a variable called where. So whatever the user types in, it will be saved in the variable called where.

So in my computer memory, the way this looks like, if the user types in right, that particular sequence of characters, that will be saved as the variable where. So then we finish this first line of the code here. And then we check while the value of where is equivalent to right, what are we going to do? We're going to ask the user again, where do you want to go? Left or right?

So I'm going to say right again. And then this memory is going to look exactly the same. If the user keeps typing in right, I keep reassigning the variable where to have the value right. At some point, the user might type in left, in which case, we're going to lose the binding from a variable, where, from the specific sequence of characters, right, we're going to bind it now to the characters left.

So at some point, after repeating this many times, the user will type in left. And we're going to have where is equal to left. And at that point, when the condition is being checked again, Python will say, nope, this is not equivalent. So I'm not going to go inside this code block. I'm just going to go down here and print "You got out of the Lost Forest."

So in code, the way this looks is-- it's this first one here. So you're in the lost forest, go left or right. So if I type in right, it just keeps asking me which way to go. And at some point, I can type in left and I'm out. So it's pretty cool, right? We just made our own level in this text adventure.

Let's have you think about this question. What if the user types in capital RIGHT? What do you think will happen? Are we going to ask the user to go left or right again? Or are we going to tell them that they got out of the forest?

STUDENT: They got out.

ANA BELL: Yeah, they got out. Do you want to say why?

STUDENT: Because it's not lowercase.

ANA BELL: Yeah, exactly, because it's not lowercase. So remember, when we're doing comparison-- so the == on strings, it has to be the same case. It's case sensitive. And so capital RIGHT or even some combination, like just Right is also going to give us that we got out. So this is counterintuitive to what we see as humans because we see right no matter what to mean right. So a workaround for this would be to use a command on the string to maybe convert everything to lowercase, just so it's more easily compared or something like that.

OK. So another use of while loops is with numbers. Let's look at this example. I'm going to ask the user for an integer. And then I'm going to do something really simple. I'm going to print x to the screen however many times the user told me. So if the user gives me 4, I'm going to print x four times to the screen.

So what is this code doing in memory? Well, the user gives me, let's say, 4. What happens step by step? First, we see our while loop. So I'm going to check whether 4, the current value of n, is greater than 0. Yes, that's true. I'm going to print x to the screen. And then I'm going to do the next line of code. That's part of this indented block, which is to take n and assign it to whatever n is minus 1. So I'm going to lose the 4. And I'm going to take 4 minus 1 to be 3, create a new object, and bind n to the 3. OK?

Next, Python-- again, it's part of a while loop. So automatically, it looks at the condition again and says, well, now, the value of n is still greater than 0. Yeah, 3 is still greater than 0. So again, we're going to lose the binding-- sorry, we're printing x to the screen first. And then we lose the binding from the current value of n, 3, to 2. So we're decrementing n by 1 each time through this while loop.

Then, again, Python checks the condition. It says 2 is still greater than 0. So again, we print another x to the screen. And then we decrement n by 1. So we're binding n to 1. Again, Python checks the condition. Is 1 still greater than 0? Yes. So we print another x to the screen. So we've printed four x's now to the screen.

And then Python says now, I'm going to make n to be 0. And then at this point, Python will do another check. It's going to say, is 0 greater than 0? And that's going to be false. So it's not going to execute the code block anymore. And the program will be done. There's no code to rejoin anymore. There's just the end of the program.

So we would have printed four x's to the screen. And this is in the Python file I gave you. You can feel free to run it to double check. My question is, what happens-- and this is a really common mistake. What happens if we forget this last line? We can try it. I can try it in here.

STUDENT: [INAUDIBLE]

ANA BELL: Yeah, exactly. It's going to go on forever. I'll show you what that looks like. So this is the code when we just have it working as usual. So if I type in 3, it prints three of those x's. But if I happen to forget to write this last line-- I'm just going to comment it out. And if I run the program, I can enter any number. And it'll just keep printing stuff to the console. It's just printing a whole bunch of stuff. So you can see this is all the stuff it printed.

So yeah, we don't have a program that terminates because the condition here is never actually being-- the variable that's part of this condition is never actually being changed inside my loop. And so that's a big problem. When that happens, what we can do-- and what I just did here is-- I'm going to-- you can click the shell and hit Control-C or Command-C on a Mac.

And that will just end the program manually. Or you can just click the red X in the corner. So here's another example of it going infinite. And there's this little-- sorry, red box in the corner. You can click that. Or you can click the three lines to interrupt kernel. All that will stop the program.

So in this class, we're not actually going to write programs that take seconds to run. So if you find yourself waiting for your program for more than one or two seconds, then likely you've entered an infinite loop. So you'll want to stop it and try to see where your program went wrong.

OK. So give this a try, if you want, just so you get the hang of stopping an infinite program because you'll very likely write a program that doesn't terminate. So while true-- what's the condition here? It's just true, right? So there's no condition that's being checked. This program will run always an infinite times no matter what. So that's just this little You Try It down here on line 44. Just run it as soon. As you run it, it's just going to print that to the screen over and over again. Be sure to click the shell to put the focus on there and hit Control-C or hit the red X.

All right, so the big idea with while loops is that they can repeat the code inside them indefinitely. So we have to be a little bit careful with what our conditions are and whether we're actually making progress towards having that condition become false at some point. And when that happens, when they run indefinitely, you'll have to manually intervene to close the program.

OK. So now that we've seen a loop with a little bit of numerical computation inside it-- so we were changing the value of n inside our loop-- let's have you work on this little program. It's an extension of the Lost Woods. This is exactly the same program that I just ran a few slides ago.

But what I want you to add is an extra printout. So when the user says right more than two times, the next time you ask them whether they go left or right, I'd like you to print a sad face right before you ask them that question. It can be on a different line. It doesn't have to be on the same line.

And the way to do that is to try to create a new variable that's going to be like your counter, that keeps track of how many times the user has-- how many times this while loop has repeated. So I'll give you a couple of moments to do that. And then we'll write it together.

As usual, the You Try It is in here. So you can just uncomment the code with-- in Spyder, its Control-1 or Command-1 to batch uncomment. And then you can work off of this code to try to improve it.

OK. So does anyone have a start for me? How can we do this? You don't have to give it to me in full. We can work our way up to the final program. But what's your first thought here? Yes? Sorry.

STUDENT: [INAUDIBLE] variable like n or something incremented every time the while loop goes through.

ANA BELL: OK. So we can create a variable, n, at the beginning of our program. What do you want to make it?

STUDENT: 0.

ANA BELL: 0, OK, good. 0 will keep track of-- or n will keep track of how many times we've gone through the loop. So inside our program, when do we want to change n? Sorry. Every time we go through the loop, right? So every time we want to go through the loop, we want to change n to be a new value. So maybe we want to increase it by 1. So n is equal to n plus 1.

So now, this will keep track of how many times we've gone through the loop. And we can actually double check this by printing n. So if we run it and we say right, we've gone once. Right, we've got twice. Right, we've gone three times. So this means we're incrementing it correctly. Now, what can I do with this variable n now that I have it and I know it's incrementing correctly? Yeah?

STUDENT: Set up an if statement [INAUDIBLE].

ANA BELL: Yep, we can set up an if statement. So we can check if that value of n is greater than 2 according to the specification here. What do you want to do when if is greater than 2? Print something, right? So we can print the sad face. And let's try to run it now. So if we immediately hit left, it still works. If we go right one time, nothing. Another time, nothing. Right a last time, sad. And from now on, it's going to keep showing me the sad face. OK. Questions about this code? Yeah?

STUDENT: Is it possible to test for something that's not equivalent. Is there a sign that's different than the two equals or not equivalent?

ANA BELL: Can we check for non-equivalency? Here?

STUDENT: Yeah.

ANA BELL: So this particular check checks for what the user typed in. It's possible we can add this if statement that checks for the n in here and something else. But then we would have to have maybe another-- I'd have to think about it. But it might be possible to try to combine them inside the while loop.

STUDENT: So there's a symbol that allows you to do that?

ANA BELL: Oh, to do not equals? That would be the not equal sign, yeah. So another thing we can do with while loops is to iterate through numbers in a sequence. If we do this, there's a really common pattern, which actually leads us to the next kind of loop we're going to see on the next slide.

But the pattern, when you want to iterate through numbers in a sequence, is you first set a loop variable before the while loop. Inside the condition for the loop, you do some sort of check with that variable. So n was my loop variable outside the loop. And then I test it inside the while loop. So n is less than 5.

And then within the while loop, you can do whatever commands you want to do with that n. But then you have to remember to change it in some way because if you don't change it in some way, this while loop condition will always be true. So here, I'm incrementing n by 1 because it starts from 0. I want n to get to something, something above 5, which will lead to my condition becoming false.

So this pattern actually exists in a bunch of different programs. So here's a program that calculates factorial for me. And here, I'm calculating 4 factorial. I'm not excited about the number 4. That's 4 factorial. How do we do this? Well, there's a lot of things I'm initializing here. But the more you work with loops, you'll get used to seeing what is the loop variable.

So i is actually going to be my loop variable. Here, it's being set to some value, initially outside the loop. Inside the conditional, I'm doing some sort of condition check with it. And then inside the body of that conditional, I'm changing it in some way. That gives me some sort of-- that takes me to the end of my conditional here. So I'm setting i to 0. I'm incrementing i by 1 each time through the loop. And I'm making forward progress towards making i greater than x, at which point my conditional will become false.

The rest of the code-- x is equal to 4-- just sets the thing I want to get the factorial of. And this factorial variable is kind of my running product. So it's the thing that I'm going to keep multiplying to figure out what the factorial is. So here, I'm initializing it to 1. And inside the loop, I'm multiplying it by my loop variable every time.

So I'm not going to do a memory diagram for this example, but I will do the Python Tutor. And I'm going to step through to show you exactly what this is doing. So x is 4. i is 1 originally. And factorial is 1. So x is the thing I want to get the factorial of. i is going to be my loop variable. And factorial is my running product.

So next step, i, 1 is less than or equal to 4. So I'm going to enter the loop. Python will calculate the factorial as whatever it is right now multiplied by 1, i. So it's still 1. And then I'm going to increment i to whatever it is from whatever it is now to 1. So I just want to mention this i += 1 is equivalent to saying i equals i plus 1.

And this is true no matter what variable you have here. Basically, if you have fact *= 2 or something like that, that basically means factorial equals fact times 2. So that's the pattern here. These are equivalent and these are equivalent. This is just shorthand notation in programming. So that's what this line here means. i += 1 means I equals I plus 1.

So at this line here, I'm taking whatever i is and adding 1 to it, 2. And then I do the check again. And remember, Python does this automatically. Because we're using a while loop, it goes back to the condition and checks it again using these new values for the variables. 2 is still less than or equal to 4. So again, we go inside the loop body. Factorial is whatever it is right now, 1, multiplied by i, 2.

i is going to be 2 plus 1, 3. And then again, I'm checking that 3 is less than or equal to 4. It still is. So then we're going to do factorial is whatever it is now, 2, multiplied by whatever i is, 3. So now, it's 6. i is going to be one more than what it is right now, 4. 4 is still less than or equal to 4. We're going to go inside the body. factorial is whatever it is right now, 6, multiplied by 4. And that's 24.

And then i is going to be whatever it is right now plus 1, 5. At this point, Python says, is 5 less than or equal to 4? No. And then it breaks the loop. And it goes to print this statement, 4 factorial is-- and then it grabs whatever the value of the factorial is. So here, I'm using this fstring print that notation that we learned about last lecture. So I encourage you to go through it yourself, just step by step. That's what Python Tutor's really, really useful for.

OK. So let's look at a different kind of loop called a for loop. And the for loop is going to allow us to rewrite that special kind of while loop that we saw where we initialize a variable, we test the variable, we do something to the variable within the code in a more efficient, more readable way.

So in terms of our Netflix example, a for loop would be equivalent to something like Netflix, if you're not interacting, with it cuts you off after four episodes to save bandwidth. And so there's a sequence of four episodes it knows it's going to go through if you're not interacting with it.

So if you've already gone through your sequence of four episodes you're allowed to watch without any interaction, it's done. It cuts you off it says, "Are you still watching?" But if there are still more episodes, if it only showed you two out of the four, then it's going to keep showing you more episodes until it's shown you the four.

So this is the program we had with while loops a couple slides ago. And remember, we are initializing a variable. We were testing the variable with some condition here. And then we were incrementing the variable or doing something that gives us nice forward progress towards making this condition false.

But it's really verbose. Certainly, it works. You can do it, but it's very easy to forget to do this, something like this, in which case you'll get an infinite loop. With a for loop, those four lines of code just look like this, these two lines of code. So if there's a sequence of values you ever want to iterate over, that's what for loops are useful for.

So the syntax for a for loop looks a little bit different than a while loop. It starts with a for keyword. This is a variable that you get to name, whatever name you'd like. The keyword in tells Python I'm going to make this variable take on values in this sequence. And again, we have a colon that tells Python we're going to start a code indentation here. And whatever lines of code you have that are indented are going to be executed however many sequence of values you have.

So the first time through the loop, Python will make this variable name take on the first value in the sequence. And then it's going to execute this code. Automatically, Python, after it finishes executing these codes, it will go back and set this variable have the next value in the sequence and execute the same lines of code. When it's done, it's going to make the variable here take on the next values in the sequence and execute those lines of code. And so these lines of code will effectively be executed however many values you have in your sequence.

So more practically speaking, here, we have a variable-- so n, in this case-- in some sequence of values. In this case, I'm saying range 5. We're going to print the value of n. So I'm going to introduce range now. Range is a way for us to grab numerical sequence of values that have some sort of pattern.

So if we just say range some number, the pattern is we start at 0 and we go up to but not including that number. So range 5 means the sequence of values Python will iterate over are 0, 1, 2, 3, and 4. Range 10 means 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. So we go up to, but not including the value in the range starting from 0. So each time through the loop, Python will change the value of n to be every one of those values automatically.

So these two lines here, for n in range 5 print n, the way they look like behind the scenes-- and Python does this for you-- is the first time it encounters for loop, it sets n to be 0. That's the first value in my sequence. And then it prints the value of n, 0.

Next time through the loop, Python will say, OK, I've done what you asked me to do inside the code loop, print n. I'm going to make n have the next value in my sequence. So it loses the binding from the 0 and makes it be 1. OK, I've made n 1. Now, what do you want me to do? Well, I'm going to execute whatever is indented, print n. So I'm going to print 1. So I've already printed 0. Then I've printed 1.

I'm finished executing the code inside the loop. So now, n is going to get the next value in the sequence. Lose the binding from 1 and you get 2 and so on and so on. And by the end, this program will have printed 0, 1, 2, 3, and 4, every single value in my range.

So it turns out that we can actually make range have three values inside the parentheses, not just one. One is shorthand notation if you ever want to start from 0 and want to go and go up to and including-- sorry, up to but not including the value in the parentheses.

But you can actually give it three values-- a start, a stop, and a step. And Python will automatically generate a sequence of values that matches this pattern. So this should seem familiar to you because we've seen something like this when we were doing strings, except that we weren't doing parentheses, we were doing square brackets. And we weren't doing commas, we were doing colons. But it's the exact same idea.

Here, we're generating numbers, actual integers that we want a loop variable to take on. So if we omit start and step-- start by default is 0 and step by default is 1. If we omit step, by default, it will be 1. So here, i in range 4, the variable i will take on the values 0, 1, 2, and 3. i in range 3, 5, i will take on the values 3 and 4. So we go up to, but not including the 5.

Think about these three questions. So what are the range of values in the first one? And what are we going to print? So in 1, 4, 1, what range of values are we going to have i be? So i is going to be what? 1, 2, 3?

STUDENT: [INAUDIBLE]

ANA BELL: Yes, and we stop. We go up to, but not including the stop, which is a 4. And what are we printing?

STUDENT: [INAUDIBLE]

ANA BELL: Yeah, 1, 2, 3. How about the next one, j? What will the values of j be?

STUDENT: [INAUDIBLE]

ANA BELL: 1, 3. And that's it, yep, because we're going every other value. And what are we printing here?

STUDENT: [INAUDIBLE]

ANA BELL: Yeah, exactly. So we're doing an operation with each one of these j's. So we're going to print 2, and then 6. And how about the last one?

STUDENT: [INAUDIBLE]

ANA BELL: We're stepping backward, right? The negative 1. So what's our start?

STUDENT: [INAUDIBLE]

ANA BELL: 4. And then?

STUDENT: 3

ANA BELL: 3, 2, 1. And that's it. We're going down to, but not including the end, right? So we're not going to include the 0. And what are we printing here?

STUDENT: [INAUDIBLE]

ANA BELL: Yes, four dollar signs for the first time, and then three dollar signs, and then two dollar signs, and then one dollar sign, exactly. So the body obviously can do operations and can manipulate that loop variable. So each time that variable goes through, it changes. And then you can use that to your advantage.

So here's another example of something useful. We can use for loops to keep track of how many times we're going through a loop. And in this particular case, we're writing a program that sums all the values from 0 all the way up to but not including whatever is in here.

So how are we doing this? Let's do the memory diagram. We've got mysum is equal to 0 as the first line. So this will be 0 in memory, bound to the name mysum. And then for loop will generate for me the values 0 through 9, including.

So i, the first time through the loop, will have a value of 0. So we're going to do the operations or the code we're asked to do when i is 0. So mysum will be whatever it is right now plus whatever i is, 0. So it stays 0. Python's done with the code inside. So now, it's going to take i and change it to the next value in the sequence, 1.

Now, we're going to do again the stuff inside the loop with i being 1. So we're going to take mysum, whatever it is right now, and add 1 to it. So it's 1. And then we're done there. So Python will take i to be the next value in the sequence, 2. And then we're going to do again mysum is whatever it is now, 1, plus whatever i is now, 2. So it's 3.

Again, i will increment to 3 automatically. That's the next value in the sequence. So mysum will get a value of 6. And then i will change to 4, so on, and so on, and so on until i becomes 8. That's sort of towards the end. When i is 8, the value of mysum is 36, right? 0 plus 0 plus 1 plus 2 all the way up to 8 is 36. And then when i becomes 9, Python will take mysum, whatever it is right now, 36, add 9 to it to give us 45.

And then that's the end of the program, right? There's nothing else to do, except to print mysum. So at the end of this loop, it's gone through 10 times adding 0 all the way up to 9. We're going to print 45. Yeah?

STUDENT: So I tried running it. It printed 45 at the top, and then just went 0, 1, 2, 3, 4 [INAUDIBLE].

ANA BELL: Oh, maybe you have another print statement. Or it might be part of another program that's being run beforehand that you didn't comment out. Yeah, next question?

STUDENT: [INAUDIBLE] the += syntax [INAUDIBLE]?

ANA BELL: The += equals what it means? Oh, it just means it would be like mysum equals mysum plus i. It's just shorthand notation because most of your variable names might be really long. And it's really annoying to retype them. And so that's generally why that shorthand notation exists, yeah. But it basically means take whatever mysum is and add i to it. And save it back into the variable mysum.

OK. Let's have you try this code real quick. So here is code. It's already on the Python file to start out with. I want you to have this code-- it's pretty close to working. But there's one issue. So we have this for loop that starts at start and ends at end. And we're keeping a running sum. And then we're printing the sum at the end.

So very similar to what we just saw, but what I want this code to do is I want it to go and sum up the start and the end. So if I have start as 3 and end is 5, I want it to add 3 plus 4 plus 5. And so this code is not doing quite that. And I would like you to fix it or to tell me how to fix it. So it's down here on line 140-ish.

First thing you should do is run it and maybe see what answer it actually gives you. So I just ran it. It gave me a 7. When you're encountering an output that's not quite what you expect, one of the first things to do-- you can obviously use the Python Tutor. But another thing you can do is put print statements at various places. Useful places would be inside for loop.

So here, we can print i. That's a reasonable thing to print out. And maybe we'll see exactly what values of i we are adding because we know the summing works. We just wrote the program on the previous line-- or previous slide. So we got 3, 4, 7, which is a little confusing. Let's make our print statement be a little bit better. i equals comma and then print the actual value of i. So what do you guys notice? i is 3. i is 4. And then it prints the sum 7. What's the problem with this code? Yeah?

STUDENT: [INAUDIBLE]

ANA BELL: Yeah, we're not adding 5, right? We're just adding 0, originally, plus 3 plus 4. And we've never added 5. So how can we fix that? Yeah?

STUDENT: end plus 1.

ANA BELL: Yeah, we can do n plus 1, exactly. So the range, remember, grabs the-- oh, plus 1. The range grabs these values as numbers that it's working with. So start is OK. end is OK. But we go up to but not including end. So if we go to end plus 1, we're going to go up to but not including end plus 1. So if we run it now, it looks much better. So we've got i is 3, i is 4, i is 5, and the sum is 12. Perfect.

So print statements are very useful when debugging code. Questions about this one? Or does this make sense? OK. The last slide I want to do before we do a summary is just to show you that factorial code we saw using a while loop a few slides back. So it looks really verbose. We kind of have to think about it for a while before we realized what it's actually doing. But it was calculating the factorial. Obviously, good variable names helped us figure that out.

That same code in a for loop looks like this-- with a for loop looks like this. So we still have the initialization of x to 4. We still initialize our factorial, our running product to a 1. But we're losing the four lines of code that make up that pattern of changing numbers with while loops into 2.

So this line, i equals 1, this while loop with the conditional, and this incrementing of i become the for loop and that's it. The for loop takes care of all of that-- the initialization, the increment, and going up to but not including the last value, x plus 1. So we're going to multiply the factorial with 1, then 2, then 3, then 4 all the way up to and including x.

So the big idea about for loops is they're going to repeat however long the sequence is. So you're able to repeat certain code a set number of times, which is really useful in some situations. While loops were useful in situations where you didn't know how many times you might want to repeat the code.

So a quick summary-- we saw some looping mechanisms today. It was a lot of syntax, I know. But the finger exercises for today will certainly help. MITx also has extra help, extra exercises. It's really important to do them, just to get the mental model for how exactly these loops work, and how they assign variables, and how they do the checks behind the scenes. And it'll help you get faster at writing code and at doing quizzes as well. So that's it for today.