PROFESSOR: All right. Let's get started, everyone. So, good afternoon. Welcome to the second lecture of 60001 and also of 600. So as always, if you'd like to follow along with the lectures, please go ahead and download the slides and the code that I'll provide at least an hour before class every day. All right.

So a quick recap of what we did last time. So last time, we talked a little bit about what a computer is. And I think the main takeaway from the last lecture is really that a computer only does what it is told, right? So it's not going to spontaneously make decisions on its own. You, as the programmer, have to tell it what you want it to do by writing programs. OK. So we talked about simple objects. And these objects were of different types. So we saw integers, floats, and Booleans. And then we did a couple of simple operations with them.

Today, we're going to look at a different-- a new type of object called a string. And then we're going to introduce some more powerful things in our programming toolbox. So we're going to look at how to branch within a program, and how to make things-- how to make the computer repeat certain tasks within our program.

All right. So let's begin by looking at strings. So strings are a new object type. We've seen so far integers, which were whole numbers, floats, which were decimal numbers, and we have seen Booleans, which were true and false. So strings are going to be sequences of characters. And these characters can be anything. They can be letters, digits, special characters, and also spaces. And you tell Python that you're talking about a string object by enclosing it in quotation marks. So in this case, I'm creating an object whose value is h-e-l-l-o space t-h-e-r-e.

And Python knows it's a string object, because we're enclosing it in quotations. They can be either double quotes or single quotes, but as long as you're consistent, it doesn't matter. And this object, we're binding it to this variable named hi. And we're using that using the equals sign, which is the assignment operator. So from now on, whenever we refer to this variable hi,

Python is going to say, oh, I know what the value is, and it's that string of characters.

So we're going to learn about two things that you can do on strings today, two operations. One is to concatenate them. And concatenation is really just a fancy word for using this plus operator, which means put the strings together. So I have this original variable named hi, and I create a new variable called name. And in it, I'm going to assign the string a-n-a to the variable name. And when I use the plus operator in between hi and name, those two variables, Python is going to look at the values of those two, and it's going to just put them together.

OK. I'm going to switch to Spider. And this is just that example from the slides. So let's see what happens. So I have the variable hi, the variable name, and I'm just concatenating those two together. And then I'm going to print that out. So if I run the code, notice it prints out "hello thereana." There's no space. And there's no space because the concatenation operator, the plus, doesn't add any spaces implicitly. So again, another example of just computer just doing what it's told. If we want to add a space, we'd have to actually insert the space manually. So that's this line here, line 8. And in this line, we're concatenating the value of the variable hi with a space. Notice we're putting it in quotation marks. Just a space. And then with name.

So if we'll go ahead and print that value, notice this was that garbage greeting there. And now we have a proper greeting, right? So that's the concatenation between strings.

And then the other thing we're going to look at related to strings is the star operator. So that's this one here on line 10. So Python allows you to use the star operator, which stands for multiplication, between a string and a number. And when you do that, Python interprets it as repeat that string that many number of times. So in this case, I'm creating a silly greeting, and I'm concatenating the value of hi, which is "hello there" with the space plus the name. So notice here, I'm using parentheses to tell Python, do this operation first, and then multiply whatever the result of this is by 3. So if I print that out, it's going to multiply the space with my name three times, and it's going to concatenate that with "hello there." So that's exactly what it printed out there.

Last lecture, we talked a little bit about print. Today, I'm going to talk about some nuances related to print. So you use print to interact with the user. It's cool to write programs that print things out to the user. So the key word here being print. And then you put parentheses after print. And in the parentheses, you put in whatever you want to show the user.

So in this little program, I have-- I created a variable named x. I assigned it the value 1, and

then I print 1. Here, I'm casting. So I'm taking the number one, the integer 1, and I'm casting it to a string. And you'll see why in a moment.

So I want to bring to your attention a couple of things here. So in the first print, I'm using commas everywhere here. And in the second print, I'm using plus. So by definition, if you-- you can use commas inside a print-- inside the parentheses of print. And if you use a comma, Python is going to automatically add a space in between the two things that the comma is in between, the values. So "my fav num is" is the first thing. And the second thing is whatever's after the comma. Let's take x. So if you use a comma, Python is going to automatically insert a space for you.

Sometimes, you might want that, sometimes you might not. If you don't want that, you can use the concatenation operation, the plus. And you can add all of your little bits together to create one big string. If you're using commas, the items, the objects in between the commas, do not all have to be strings. That's the plus side of using commas. But the downside is you get spaces everywhere. If you use plus operator, the plus side is Python does exactly what you tell it to do, but everything has to be a string object. So "my fav num is" is a string object. You have to convert all of your numbers to string objects, and so on.

So if we look at Spider-- This is the same-- almost the same code. So here, I don't have spaces anywhere. So you can see that the first line here has commas everywhere. So I'm going to have spaces in between every one of the things that I'm printing out. This line here is sort of a combination between commas and concatenation. So depending on where I used the comma, I'm going to have an extra space. And this line here just has concatenation everywhere. So if I run this, notice this very first line added spaces everywhere in between all my objects. The second one added spaces somewhere. And you can sort of trace through and see exactly where the spaces were added. And the last line here didn't add spaces anywhere.

So printing things out to the console is nice, but the second part of sort of writing an interactive program is getting input from the user. And that's the more interesting part. So if you've done problem set 0, you might have sort of already tried to understand this on your own. But here we are. So the way you get input from the user is using this command function called input. And inside the parentheses, you type in whatever you'd like to prompt the user with. So in this case, in my example here, I have input, and then here I said "type anything." So the user is going to see this text here, and then the program is just going to stop. And it's going to wait for the user to type in something and hit Enter. As soon as the user types in Enter, whatever the

user types in becomes a string. If a user types in a number, for example, that becomes the string of that number. So everything the user types in is going to be made as a string.

In this line right here, whatever these the user types in becomes a string. And we're going to bind that string object to this variable named text. So now, further in my program, I could do whatever I want with this variable text. In this case, I'm going to print 5*text. OK. So if the user, for example, gave me "ha," I'm going to print "ha" 5 times. If the user gave me 5, what do you think the user is-- what do you think is going to be printed out? 25 or 5 five times? Great. Yes. Exactly. 5 five times.

Oftentimes, you don't want to work with numbers as strings, right? You want to work with numbers as numbers, right? So you have to cast. And we learned that last lecture. You cast by just putting in this little bit right in front of the input. And you can cast it to whatever type you want. Here I cast it to an int, but you can also cast to a float if you want to work with floats. And that converts whatever the user typed in, as long as it's some number that Python knows how to convert, into the number itself. So in this case, if the user gives me 5, I'm going to print out 5 times 5 instead of 5 five times. So that's the code here.

So the first bit is I'm going to get the user to type in anything, and I'm going to put 555. And then when I type in the number, since I'm casting it, I'm going to do operations with the number. Yeah, question.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Why do you want to cast to-- oh. The question is why do you want to cast to a string? Why do you want to cast a string to a number?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Oh, so Python always-- whatever you type in, just by default, by definition of the input command, Python always makes it a string. So if you want to work with numbers, you have to explicitly tell it, I'm going to work with a number. So even if you give it the number 5, it's going to think it's the string 5. Yeah. That's just how input works.

The next thing we're going to look at is ways that you can start adding tests in your code. And before you can start adding tests in your code, you need to be able to do the actual tests. So this is where comparison operators come in. So here, let's assume that i and j are variables.

The following comparisons are going to give you a Boolean. So it's either going to say, this is true or this is false. So that's going to be your test.

So if i and j are variables, you're allowed to compare ints with ints, floats with floats, strings with strings. And you're allowed to compare ints and floats between themselves, but you're not allowed to compare a string with a number. In fact, if you even try to do that in Python-- in Spider here, if I try to say, is the letter a greater than 5? I get some angry text right here. And this just tells me Python doesn't understand the meaning of-- how do I compare a string with a number?

OK. So just like in math, we can do these usual comparisons. We can say if something is greater than something, greater or equal to, less than, less than or equal to. I'd like to bring to your attention the equality. So the single equals sign is an assignment. So you're taking a value, and you're assigning it to a variable. But when you're doing the double equals sign, this is the test for equality. Is the value of variable i the same as the value of the variable j? And that's, again, also going to give you a Boolean either true or false. And you can also test for inequality with the exclamation equal. So that means, is the value of the variable i not equal to the value of the variable j? True if yes, false if no.

OK. So those are comparison operators on integer, floats, and strings. On Booleans, you can do some logic operators. And the simplest is just inverting. So if a is a variable that has a Boolean value, not a is just going to invert it. So if a is true, then not a is false, and vice versa. This is a table that sort of represents what I've said here. So you can do-- you can use and and or. These are key words in Python. You can use those two key words on variables, on Boolean variables. And you get the result a and b is only true if both a and b are true. And a or b is only false if a and b are false. And this is the complete table just in case you need to reference it.

All right. So now that we have ways to do logical-- question right there.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, great question. So what does it mean to compare a string with a string with the greater than? So that's just going to compare them, lexicographically. So does it come first in the alphabet? So we can even test that out. We can say, is a greater than b? And it's false. So b comes later in the alphabet than a.

OK. So now we have ways to do the tests. So we can add some branching to our programming toolbox now that we have ways to do tests. This is a map of MIT. I'm going to go through sort of a little example to motivate why we would want to do branching in our code. And I think after this lecture, you'll be able to sort of code up this algorithm that I'm going to explain. So most of us see MIT as a maze. I first did when I came here. When I first came here, obviously, I signed up for the free food mailing list. And MIT, being a maze, I had no idea where to go, what the shortest path was to free food. So one way to think about it is all I wanted to do was get to the free food.

A very simple algorithm to get there would be to say, OK, I'm going take my right hand, and I'm going to make sure that my right hand is always on a wall. And I'm going to go around campus with my right hand always being at a wall. And eventually, I'll get to where the free food is. There might not be any left, right? But I'll be there. So the algorithm is as follows. If my right hand always has to be on a wall, then I'm going to say, if there's no wall to my right side, then I'm going to go right until I get to a wall. Then if there's a wall to my right, and I can go forward, I'm just going to keep going forward. If I keep going forward, and there's a wall to my right and in front of me, I'm going to turn around and go left. And then if there's a wall to my right, in front of me, and to the left, then I'm going to turn around and go back.

So with this fairly simple algorithm, I just follow the path always keeping the wall to my right. And eventually, I would end up where I need to be. So notice, I used, just in plain English, a few key words. If, otherwise, things like that. So in programming, we have those same constructs. And those same sort of intuitive words can be used to tell Python to do something or to do something else or to choose from a different set of possibilities. And this way, we can get the computer to make decisions for us. And you might be thinking, well, you said that computers can't make decisions on their own. It's not. You, as programmers, are going to build these decisions into the program, and all the computer is going to do is going to reach the decision point and say, OK, this is a decision point, should I go left or should I go right? Or which one do I pick? And these sort of decisions are created by you as a programmer. And the computer just has to make the decision and choose a path.

OK. So in programming, there's three sort of simple ways that you can add control flow to your programs. And that's making one decision and choosing whether to execute something or execute something else. The first is a simple if. And given a program that just linearly has statements that get executed, whenever I reach an if statement, you're going to check the

condition. The condition is going to be something that's going to get evaluated to either true or false.

So I've reached the condition here. And if the condition is true, then I'm going to additionally execute this extra set of expressions. But if the condition is false, then I'm just going to keep going through the program and not execute that extra set of instructions. How does Python know which instructions to execute? They're going to be inside this what we call code block. And the code block is denoted by indentation. So it's going to be everything that's indented is part of that if code block. Typically, four spaces is indentation.

OK. So that's how you write code that decides whether to execute this extra thing or not. Now let's say I don't just want to execute an extra thing, I want to reach a point where I say, I'll either go down this path or I'll do something else. That's this right here. So this if else construct says this is my code, I've reached my decision point here, if the condition inside the if is true, then I'm going to execute maybe this set of statements here. But if the condition is not true, then I'm not going to execute that set of statements, and instead I'm going to execute under whatever else is. So using this construct, I'm either going to do one set of expressions or the other, but never both. And after I've executed one or the other, I'm going to continue on with just the regular execution of the program.

OK. So we're able to either choose one thing, choose one thing or another, but what if we want to have more than one choice? So if some number is equal to zero, I want to do this. If it's equal to 1, I want to do this. If it's equal to 2, I want to do this, and so on. That's where this last one comes in. And we introduced this other key word here called elif. So that stands for short form for else if. So first we check if this condition is true. So we're going through our program, we've reached our decision point, if the condition is true, we're going to execute maybe this set of instructions. If the condition is not true, maybe we'll check-- if the condition is not true, we will check this next condition. That's part of the elif right here. And if that one's true, we're going to execute a different set of instructions. You can have more than one elif. And depending on which one's true, you're going to execute a different set of instructions. And then this last else is sort of a catch all where if none of the previous conditions were true, then just do this last set of expressions.

So in this case, you're going to choose between one of these three-- one of these four roots, or however many you have. And then when you're done making your choice, you're going to execute the remaining set of instructions. So the way this works is if more than one condition is

true, you're actually just going to enter one of them. And you're going to enter the very first one that's true. So you're never going to enter more than one of these code blocks. You always enter one, and you enter the first one that evaluates to true.

So notice that we denoted code blocks using indentation. And that's actually one of the things that I really like about Python. It sort of forces you to write pretty code and nice looking code and just code that's very readable. And that forces you to indent everything that's a code block. So you can easily see sort of where the flow of control is and where decision making points are and things like that. So in this particular example, we have one if statement here, and it checks if two variables are equal. And we have an if, elif, else. And in this example, we're going to enter either this code block or this one or this one, depending on the variables of x and y. And we're only going into one code block. And we'll enter the first one that's true.

Notice you can have nested conditionals. So inside this first if, we have another if here. And this inner if is only going to be checked when we enter the first-- this outter if. I do want to make one point, though. So sometimes, you might forget to do the double equals sign when you are checking for equality, and that's OK. If you just use one equals sign, Python's going to give you an error. And it's going to say syntax error, and it's going to highlight this line. And then you're going to know that there's a mistake there. And you should be using equality, because it doesn't make sense to be using-- to assign-- to be making an assignment inside the if.

So we've learned about branching. And we know about conditionals. Let's try to apply this to a little game. And spoiler, we won't be able to. We'll have to learn about a new thing. But back in the 1980s, there was the Legend of Zelda-- cool graphics-- where there was a scene with the lost woods. Oversimplification if anyone's a Zelda die hard fan. But the basic idea was if you entered the woods, you entered from the left to the right. And then as long as you kept going right, it would show you the same screen over and over again. And the trick was you just had to go backward, and then you'd exit the woods. So very simple.

Using what we know so far, we could sort of code this up. And we'd say something like this. If the user exits right, then set the background to the woods background. Otherwise, set the background to the exit background. Now let's say the user-- and then in the else, we're done. Let's say the user went right. Well, you'd show them the woods background, and now ask them again, where do they want to go? If they exit right, set the background to the woods background. Otherwise, set the background to the exit background, and so on.

So you notice that there's sort of no end to this, right? How many times-- do you know how many times the user might keep going right? They might be really persistent, right? And they'll be like maybe if I go 1,000 times, I'll get out of the woods. Maybe 1,001? Maybe. So this would probably be-- who knows how deep? These nested ifs. So we don't know. So with what we know so far, we can't really code this cute little game.

But enter loops. And specifically, a while loop. So this code here that could be infinitely number of nested if statements deep can be rewritten using these three lines. So we say while the user exits right, set the background to the woods background. And with a while loop, it's going to do what we tell it to do inside the loop, and then it's going to check the condition again, and then it's going to do what we say it should do inside the code block, and it's going to check the condition again. And then when the condition-- as long as a condition is true, it's going to keep doing that little loop there. And as soon as the condition becomes false, it's going to stop doing the loop and do whatever's right after the while.

OK. So that's basically how a while loop works. We have while. That's the key word. The condition is something that gets evaluated to true or false. And once again, we have a code block that's indented, and it tells Python, these are the expressions I want to do as long as the condition is true. So the condition is true, you evaluate every expression in the code block. When you reach the end of the expression-- end of the code block, you check the condition again. If it's true still, you keep doing the expressions. Check it again, and so on.

So here's a little game. And with these lines of code, we were able-- we can code up the lost woods of Zelda. Even worse graphics, by the way than the original Zelda is this one that I coded up here. So I print out the following things. "You're in the Lost Forest. Go left or right." And my program's going to say, "You're in the Lost Forest. Go left or right." It's going to get user input. It's going to say while the user keeps typing in right, show them this text, and ask them again. So I'm asking them again by just saying input here again. And that's it. That's going to just keep getting input from the user. And if the user doesn't type in right, and maybe types in left, you're going to exit out of this loop, and print out, "You've got out of the Lost Forest."

So I have to show you this, because I spent too much time on it. But I decided to improve on the code that's in the slides. And I've written here ways that you guys can also improve it. So if I run my code-- "You're in the Lost Forest. Go left or right." So if I say left, then yay, I got out of

the Lost Forest. But if I go right, then I'm stuck, right? I took down some trees. You can see there's no more trees here. I made a table, and then I flipped it over.

So the expansion to this if you want to try it out-- I put this in the comments here-- is try to use a counter. If the user types in right the first two times, just make that a sad face. But if the user types in more than two times, make them cut down some trees and build a table and flip it. That's a cute little expansion if you want to test yourself to make sure you are getting loops.

OK. So so far, we've used while loops to ask for user input. And that's actually somewhere where it makes sense to use while loops, because you don't actually know how many times the user is going to type in something. You can use while loops to keep sort of a counter and to write code that counts something. If you do that, though, there's two things you need to take care of. The first is the first line here, which is sort of an initialization of this loop counter. And the second is this line here, which is incrementing your loop counter.

The reason why the second one is important is because-- let's look at our condition here. So while n is less than five. If you didn't have this line here, you would never increment n. So every time through the loop, you just keep printing zeros. And you would have an infinite loop. I do want to show, though, what-- if you do have an infinite loop, it's not the end of the world. So I can say something like-- so while true, print zero. So this is going to give me an infinite loop in my program. And-- whoop. OK. So notice it's just printing the letter p over and over again. And if I let it go any longer, it's going to slow down the computer. So I'm going to hit Control-C or Command-C maybe. And it's going to stop the program from printing. So just in case you ever enter infinite loops in your programs, just go to the console and hit Control-C, and that's going to stop it from sort of slowing down the computer.

OK. So going back to this example, I was saying that if you're using counters-- variables in order to sort of count up inside the while loop, you have to take care to initialize a counter variable first. And then to increment it, otherwise you'll enter an infinite loop. That feels a little bit tedious. And so there's a shortcut for doing that exact same thing.

So these four lines, you can rewrite those into these two lines right here using this new type of loop called a for loop. So the for loop says, for some loop variable-- in this case, I named it n. You can name it whatever you want. In range 5-- we're going to come back to what range means in a little bit-- print n. So every time through the loop, you're going to print out what the value of n is. Range 5 actually creates internally a sequence of numbers starting from 0 and

going to that number 5 minus 1.

So the sequence is going to be 0, 1, 2, 3, and 4. The first time through the loop, you're going to say n is equal to 0. Or internally, this is what happens. N gets the value of 0. You're going to print n. Then you're going to go back to the top. N gets the value 1. Then you're going to go execute whatever is inside. So you're going to print 1. Then you're going to increment that to the next value in the sequence. You're going to print out 2, and so on.

So this is the general look of a for loop. So we have for some loop variable-- again, can be named whatever you want-- in range some number. Do a bunch of stuff. And again, these are part of this for loop code block. So you should indent them to tell Python that these are the things that you should do. So when you're using range some number, you start out with variable getting the value 0. With variable having value 0, you're going to execute all of these expressions. After all the expressions in the code block are done, you're going to go on to the next value. So 1. You're going to execute all these expressions with the variable being value 1, and then so on and so on until you go to some num minus 1.

That-- so using range in that way is a little bit constraining, because you're always going to get values starting from 0 and ending at some num minus 1, whatever is in the parentheses in range. Sometimes you might want to write programs that maybe start at a custom value. Don't start at 0. Maybe they start at 5. Maybe they start at minus 10. And sometimes you might want to write programs that don't go with-- don't expect the numbers by 1, but maybe skip every other number, go every two numbers, or every three numbers, and so on.

So you can customize range to your needs. The one thing you do need to give it is the stop. So if you give it only one value in the parentheses that stands for stop. And by default, start is going to have the value 0, and step is going to have the value 1. If you give it two things in the parentheses, you're giving it start and stop. So the first being start, the second being stop. And step gets this value of 1 by default. And if you give it three things in the parentheses, you're giving it start, stop, and step in that order. And you're always going to start at the start value and stop at-- or so you're going to start at the start value, and you're going to go until stop minus 1. So those are the sequences of numbers.

So in this first code right here, my sum is going to get the value 0. And you're going to have a for loop. We're going to start from 7, because we're giving it two numbers. And when you give it two numbers, it represents start and stop with step being 1. So we're starting at 7. If step is

1, the next value is 8. What's the value after that? If we're incrementing by 1? 9. And since we're going until stop minus 1, we're not actually going to pick up on 10. So this loop variable, i, the very first time through the loop is going to have the value 7. So my sum is going to be 0 plus 7. That's everything that's inside the code block.

The next time through the loop, i gets the value 8. So inside the for loop, my sum gets whatever the previous value was, which was 7, plus 8. OK. The next time through the loop, my sum get the value 7 plus 8 plus 9. Obviously, replacing that with the previous value. So 15. Since we're not going through 10, that's where we stop. And we're going to print out my sum, which is going to be the value of 7 plus 8 plus 9. Yeah? OK. Yeah.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Do they have to be integers? That's a great question. We can try that out. I'm not actually sure right off the top of my head. So you can go on Spider and say-- let's say in this example here.

So we can say 7.1, 10.3-- yeah. So they have to be integers. OK. So that's that example. And let's erase that. In this particular example, we have start, stop, and step. And here, we're going every other value. So we're starting at 5. Tell me what the next value is supposed to be. If we're taking every other one. 7, and then 9, and then-- are we doing 11 or not? Excellent. Nice. Yeah. So we're going to the end minus 1. OK.

So it's possible that sometimes you write code where you might want to exit out of the loop early. You don't want to go through all of the sequences of your numbers. Maybe there's a condition inside there where you just want to exit the loop early. Inside the while loop, maybe you want to exit the loop before the condition becomes false. So that's where the break statement comes in. So the break works like this. It's going to-- as soon as Python sees this break statement, it's going to say, OK, I'm going to look at whatever loop I'm currently in. I'm not evaluating any expression after it that comes within my loop. And I'm going to immediately exit the loop. So I'm going inside this while, this while, I'm evaluating this one expression, and I suddenly see a break.

Expression b does not get evaluated. And break is going to immediately exit out of the innermost loop that it's in. So this while loop that has condition 2, that's the innermost loop that the break is found in. So we're going to exit out of this inner most loop here. And we're evaluating expression c. And notice, we're evaluating expression c, because it's-- expression c is part of the outer while loop. It's at the same level as this one. And these ones are part of the

inner while loop.

OK. Last thing I want to say is just a little bit of a comparison between for and while loops. So when would you use one or the other. This might be useful in your problem sets. So for loops you usually use when you know the number of iterations. While loops are very useful when, for example, you're getting user input, and user input is unpredictable. You don't know how many times they're going to do a certain task. For both for and while loops, you can end out of the loop early using the break. The for loop uses this counter. It's inherent inside the for loop. A while loop you can use a counter in order-- you can use a while loop to count things. But you must initialize the counter before the while loop. And you have to remember to increment it within the loop. Otherwise, you maybe lead to an infinite loop. We've seen as the very first example of a for loop that the while-- the for loop could be rewritten as a while loop, but the vice versa is not necessarily true. And the counterexample to that is just user input. So you might not know how many times you might do a certain task. All right. Great. That's all for today.