[SQUEAKING]

[RUSTLING]

[CLICKING]

ANA BELL: So let's start today's lecture. We're going to be looking at three different topics. The first is we're going to look at a new object type called a string. We briefly mentioned this word last lecture. Then we're going to see how we can write programs that start to get input from the user and show the user output.

And finally, we're going to go into writing a little bit more interesting programs that make decisions based on decisions that we actually input in the code, so not decision spontaneously but things that we will code within our programs.

But before we go on to these topics, I just wanted to do a quick recap of what we learned last lecture, just to make sure we're all on the same page. So we introduced the idea of an object. And every object in Python has a specific type. And the type tells Python the kinds of things you're allowed to do with that object.

We talked about, once you have objects, you can actually assign these objects to variables. And these variables basically bind a name to the object in memory. With objects, you can also create expressions by combining objects together.

And the expressions can either be things that we've seen in math, like with parentheses and with object operator object. Or they can be things like this, which was introduced in programming. It's an expression, but it's a different one. It's more like a command. Or I'm asking Python to do this operation for me. What is the object that comes back from this operation?

So today, I'm going to go over this little memory diagram we started drawing last lecture. And I'm going to use this memory diagram today as well. Here's some lines of code that we wrote last lecture. So we created-- we wrote a line in Python that created an object. Its value is 3.14, a float in memory. And the name we gave this object was pi, so just the name pi, P-I.

Radius = 2.2 is another assignment statement in Python. And it binds the name radius to the value 2.2 in memory. And once we've created these variables, we can just invoke their names. We can use their names to tell Python to grab for me the values from memory. So when Python sees pi times radius**2, that means take pi, multiply it with the radius squared.

So behind the scenes, Python goes, grabs the value 3.14 from memory, replaces pi with that value, grabs 2.2 from memory, replaces radius with that value, does the operation according to the precedence rules. And then that expression, the thing on the right hand side of this equal sign, becomes a value. That value is then created as a new object in memory right here. And that object in memory is then bound to the name area. That's exactly what this assignment states.

And we can do something like this in Python, which we can't actually do in math. If we did this in math, the expression would basically say 0 = 1. But in Python, it's totally fine because again, we evaluate the thing on the right hand side of the equal sign. So on the right hand side of the equal sign, we say I want to grab the value of radius, so 2.2, add 1 to it, 3.2, create this object in memory, here I have a whole new object in memory 3.2, and then assign it to the name radius.

So I've lost the binding from the original 2.2 and rebound the name to 3.2. So we're not modifying objects in memory. We're creating new objects in memory whenever we do such operations. We're going to see how we can modify objects way into the future.

And just for completion, when we have a line that says var = type (5*4), Python also sees this as an expression. And so as an expression, it has a value. So the right hand side of this equal sign says, well, I'm going to systematically evaluate this and say, what's 5 times 4? It's 20. What's the type of 20? It's an integer.

And so that's what the right hand side evaluates, to an integer. And I'm going to bind that value, int, to the name var. So var is another variable name. And its value is int, the type of my object, which is a little strange. So far, we've just put numbers in our memory. But we can put any object type in memory.

OK, so let's move on a little bit onto the new object type called a string. So a string is actually a sequence of case sensitive characters. The characters can be anything. We have lowercase letters, uppercase letters, the numbers on your keyboard, the special characters you see on the keyboard. Even the Enter when you do a new line or a tab has a special character associated with it.

And the way we tell Python we're creating an object of type string is by enclosing the characters we want to be part of that object in these quotation marks. So when Python sees the quotation mark, it knows you're creating a string object. So here, I'm creating the string object which has the lowercase letter m, lowercase letter e. And here, I'm creating the string object which has the lowercase y, lowercase o, and lowercase u letters.

And these objects are things in Python. And we're just going to give them a handle, a binding with some more convenient variable names, a and z. So in memory, the way this would look in our little memory diagram is we would have the string characters "me" bound to the variable a. So basically what we've seen before.

All right. So now, what are some things we can do with strings? Well, some really common operations are that we can concatenate strings or we can repeat strings. So here, I'm not going to put the z in memory. You can imagine how that would look like. But let's say I create now a variable b equals the letters myself.

What if I do a plus operator between these two strings? The plus operator tells Python that I'm going to take these two strings, the individual characters in each string, and just put them together to make one new object that is all of these letters put together. So c = a + b is another assignment operator.

And on the right hand side, we have an expression, plus operator between two objects. It's going to put me, which is the c letters, and the myself, the b letters, all together to create a new object, which I then give a handle or a binding, c. So from now on, anytime I want this particular string of characters, myself, I can just invoke the name c in my program. That's just the variable name that I gave it.

Now, notice it didn't insert a space. It didn't do me space myself because we didn't tell it to do a space. So if we wanted to do a space, we'd have to put it in ourselves. So we can concatenate-- so we can have a larger expression where we concatenate a with a space and with b together. So that will give me an entirely new object in memory, the string me space myself. This new object is bound to the name d. Is that OK so far? Does that make sense? OK.

All right. So that's concatenation. It basically takes these two string characters, puts them together in a new object. What about the star? I briefly talked about this as repeating something last lecture. Well, the star operator works between a string and a number.

It doesn't work between a string and a string. It doesn't work between things like that. It works between a string and a number in either order, so a number string or a string times number.

So here, again, it's an assignment operator. The right hand side, we're going to figure out what it evaluates to first. So a times 3 means I'm going to repeat this particular sequence of characters, me, because that's what a is, it's me, three times. So this line of code here is going to create me, me, me as a new object.

And the equal sign tells it to bind it to the name silly. So any time I want to grab this particular string of characters from memory, I can just type in silly in my program and that will automatically grab that particular sequence of characters from memory.

All right. Let's do a quick exercise to make sure that you all have this. And as you're thinking about what this does, I can start typing it in to the console. Or you can either even type it in to check yourself. So b is going to be colon. And c is going to be the close parentheses. So if we go here, we have b is equal to just this colon. And c is equal to the close parentheses.

And we don't have to do all the operations at once. We can just do something like 2 times c and figure out what that is. It's just repeating the close parentheses twice. And then we can do b plus 2 times c to give us colon close parenthesis, close parenthesis. So super happy.

What about the next one? f is a. g is actually the space b. So there's a space character in there, a little tricky. And h is 3. Is this the number 3 or the string 3 for h? Yeah, exactly. It's the string 3. So what is f plus g? Again, we can do it in pieces. What is f plus g? a space b, exactly right. a is by itself. And g is space b.

What is int h? Yeah, it's just 3. What's the type of 3? int, exactly. I just cast it to an int. So if I have f = a, g = space b, and h is equal to the string 3, f plus g, we're doing it a little bit at a time, is a space b. And h is just see the string 3 here. So if I cast it to an integer, it gives me just the number 3.

And we can wrap each of these in a type command to see the exact type. But we can tell right away. So if we do f + g multiplied by int of 3-- or sorry, oops, int of h, which is just a 3, it's going to repeat a space b three times. So there's one, there's two, and there's three.

What would have happened if I forgot to cast it to an integer? What do you think, if I just did h? An error, yeah, exactly. They're not scary. They're kind of informative once you get to know them. So it's a type error. Something's wrong with our types. Can't multiply a sequence, so an integer-- or a string, a sequence, by a non-integer. All right.

OK. So what are some other operations we can do with strings? There are some different things we can do with strings that we actually haven't seen with numbers in the last lecture. One of the more common operations is to get the length of a string. So this tells us how many characters are in the string.

So if we say s is equal to abc here, I'm creating a string with characters a, b, and c, and I'm giving it the name s. Now any time in my program when I say s, Python will replace that with this string of characters abc.

I can wrap s in this lean len command. And this len command is an expression. Basically, Python reads this, and it evaluates it to one value, so replaces this expression with one value. How many characters are in my string? So len s basically become the number 3.

So in my program, I can say something like another assignment statement down here, chars is equal to len s. This is an expression like in the previous line that evaluates to 3. So basically, this line says chars is equal to 3.

OK. That's a pretty simple operation with strings. Now we're going to get into a little bit more detailed ones that requires you to remember this Python syntax. So one thing we can do with strings is we can grab individual characters at different positions. So that's called slicing.

The syntax, or the way that we actually do this in Python, is using square brackets. So you can see this here. We have some square bracket notation and this is just how Python does it. So if we have string s is equal to the characters abc, the way we tell Python we'd like to extract a character at a particular position is by indexing into that string.

Now, in Python and modern programming languages, indexing happens from 0. We count from 0 in programming, in computer science. So what that means is the index of the first character, the index of a is 0, the index of b is 1, and the index of c is 2.

So we can say, what's the character at the first position or the first location? But in computer science speak, we say that's the character at index 0. The character at index 1 is the character at location 2 and so on. So when we're indexing into a string, we're always starting to count from 0.

So s at index 0, that's how we call this line here. It's another Python expression. It just looks different than the expressions we've seen so far. But this entire expression Python evaluates to a particular value. And the value it evaluates it to is the character at that index.

So just to show you what that looks like in here, if s is equal to abc, all we would type is this, s at index 0. And this expression evaluates to that single character, a. s at index 1, b, s at index 2, c.

S at index 3 basically says, what is the character at the fourth position? Well, abc only has three positions. So this will actually give us our second error of the class, an index error. This is a pretty common error as we start working with more complex programs. It basically means you've indexed too far into the list, either to the right or to the left.

You can index into a list with negative indices as well. So if you ever want to grab the character at the last position, so at the rightmost place, that's the character at index negative 1. It's a really convenient way to grab that last character.

You basically ask, what's s at negative 1? And Python automatically grabs for us that last character. So we don't have to use an expression like len of s negative 1. That would be c as well.

And here, I've inserted an expression, len s minus 1, directly in that index. And that's totally fine. Again, Python evaluates things in to out, left to right. It evaluated len s minus 1 to be 2. And basically, this line became what's s at index 2, which we knew was c.

AUDIENCE: So why did it make negative 1 is c?

ANA BELL: Because when you index into negative numbers, we start counting from the right hand side, just Python convention. And so s at negative 4 will give us an error as well, because now we're indexing too far to the left. There's nothing there.

OK. So we can index to get single characters. That's fine. We just use square brackets and say the index that we'd like to get the character at. We can also slice to get substrings. So instead of getting single characters, we can ask Python to get us a substring starting from one index, going up to some other index, and potentially skipping characters. You can take every character along the way. You can skip every other character or some other pattern like that.

The syntax for that is similar to slicing to get a single character, slightly different, though. It's similar in that we have square brackets involved. Slightly different because now we have to give three numbers within those square brackets separated by colons.

The first number will represent, what's the start index? So where do you want to start your substring from? The second is, what's the stop index? So we're going to take every character from that start index, going all the way up to but not including the stop.

And the step means, how many characters do we skip? So if the step is 1, we're taking every character. If the step is 2, we'll take every other character. If the step is 3, we skip every two characters and so on.

Now, there's a bunch of combinations we can do with these three numbers within the square brackets that, as you work with these exercises, you'll get the hang of. For now, it won't hurt to always give it a start, a stop, and a step. That's perfectly fine.

But something that's really common, if you're always going to take every single character, is to just omit the step part. So if you just give it two numbers, number, colon, number, Python automatically knows that your step will be 1 by default. So you're not skipping anything. If you're just giving it one number, no colons, we're back to the previous slide, where we're just grabbing one element.

And I know this is going to be a little confusing. We're going to look at an example on the next slide. But this is something you'll just have to practice a little bit in the shell with the following example, hopefully, just when you go home, just to make sure that you understand what it's doing. If you have a question, like what if I put in this number or this number? just put it in the shell and see what happens.

So let's take a look at a couple of examples. So how do we slice to get substrings? Let's say our string is this longer thing, abcdefgh. When we slice, the first thing we want to look at is the step.

Is it positive or negative? If it's positive-- and remember, if you omit it by default, it's plus 1. So if it's positive, we're going to work our way left to right, the way we read. If it's negative, we're going to work our way right to left.

So what if we index s from 3, colon, 6? The step is positive 1, so we're going to work our way left to right. So that means we're going to start at index 3, so that's the d. So we're going to grab the D and we're going to go up to-- get the substring from d up to but not including the character at index 6, the h-- the g. Sorry, the g.

So the characters we're going to grab are the d, the e, and the f. We start at 3, we go up to but not including 6, taking every character because the step is 1. What if the step was 2? So same thing as we just did, except the step is 2.

Again, the step is positive, so we're going to work our way left to right. We're going to start at index 3, so we're going to grab the d and we're going to create a new object, which is going to be the characters d. We're going to skip the e because the step is 2. Take the f, and that's it. We've worked our way up to, but not including the element at index 6.

There are some other things-- I guess tricks are-- you might want to call them that you can do. So if we just put an empty colon here, that says just make the same object again. So that will evaluate to just abcdefgh again. If we do colon, colon, negative 1, this is shorthand notation for basically grabbing for me the string backward, so hgfedcba. Just make the same string as the original one, but backward.

And we can do something like this for 1 with a step negative 2. Now the step is negative. So that means we're going to work our way right to left. We're going to start at index 4. So we're going to grab the e, we're going to skip every other character. So we're not going to take the d, but we will take the c, and we're going to go down to, but not including the character at index 1. So we're going to stop right here.

So the characters we took in this order were e and then c. So this entire expression evaluates to ec. Yes, question?

AUDIENCE: Why did you skip d?

ANA BELL: Why do we skip d? Because the step is 2. So when the step is 1, we take every character. If the step is 2, skip every other one. Yeah?

AUDIENCE: For the first one, s 3 to 6, why is g not included?

ANA BELL: g is not included just by definition. We go up to but not including the stop. So we'll go up to, but not including the character at index 6. That's just the definition of slicing in Python.

AUDIENCE: Like a character that starts up to and including stop minus 1 [INAUDIBLE].

ANA BELL: So up to and including stop minus 1 means we go up to and including 5. Right, yeah, exactly. Yep. OK. So again, if you're unsure of what a command does, always try-- you can always try it in your console, the shell, and here's an opportunity to do that. So here's a string s, ABC d3f, and I'm actually going to write this one down. Just ABC, space-- there's a space here, 3d-- what do I do? Oh, d3f. And then another space, and ghi.

So what do you guys think the first one will be? 3, colon, len s minus 1. I'll even do the indices here for you. What's the start? Yeah, the space, exactly. So it's going to be a little space. What is len minus 1?

AUDIENCE: That's the length, right? It's not [INAUDIBLE].

ANA BELL: Yeah, what's the length-- how many characters are in here? 11, yep. And minus 1 is 10. So when we do this-- when the stop is 10, that means we're going to go up to, but not including the character at 10. So we're going to go up to this h. So we're going to take the space, d3f, space gh, and we stop. Yeah.

AUDIENCE: Why do we start the next thing at 0 again? It's just--

ANA BELL: Convention.

AUDIENCE: OK.

ANA BELL: Computer science. Programming. Except for MATLAB, I think they still start indexing at 1. Other questions about this one? Is that all right? OK, how about the next one? s 4, colon, 0, colon, negative 1. What's the element at index 4? The d, yep. So we're going to grab the d. Are we working our way right or-- to the right or to the left?

AUDIENCE: Left.

ANA BELL: Yeah, exactly. So we're going to go up to, but not including the character at index 0. So we're going to get the d, the space, the c, the B. Am I taking the a? No, exactly. So that's it. D, space, CB. Yes?

AUDIENCE: If we want to include the A, would the second value be negative 1?

ANA BELL: If you did want to include the A, actually, you would want to do something different, I think. You can't go to negative 1 because negative 1 is actually this right here. That's a good question. I'd have to try it out, play around with it. But if you want to include it, I think maybe you would just do an empty-- sorry, go ahead. You would just probably do an empty colon, and by default, that means the beginning and the end. But I'd have to try it out, yeah.

How about the last one? 6, colon, 3. What's the element at index 6? The empty colon work-- OK, perfect. Thanks for trying it out. The empty colon works, yeah. If you wanted to grab B. All right, so s 6, colon, 3, what's the element at index 6?

AUDIENCE: The F.

ANA BELL: The F. OK, great. And we're working our way to the right or to the left? To the right. OK. So we're going to start here, but we need to go this way. But what's the stop index? Yeah. It's not there, it's behind us. So this last one is actually an empty string. And I'll even-- we can even-- we can try it with something else, too.

So if we have this ABC-- if I'm indexing starting from 2 and I'm going backward to 0, then that gives me the empty string. And the empty string is just quote, quote with nothing inside. So that means we didn't take any characters in that particular case. Is that all right?

AUDIENCE: Is it valid?

ANA BELL: It's valid. We just-- there are no characters in between the f and behind the f. Yeah. OK, so I'll mention the strings are actually immutable objects.

And really, a lot of the objects we've seen so far are immutable. That means they can't be modified once they're created. We've seen this already. When I draw the memory diagrams, when I create a new object, which is, for example, what's the string version of this integer or when I cast a float to an int, things like that, I'm not changing those original objects I've created, I'm just making a new green box in my memory and reassigning the name.

And we're going to see later on in this course mutable objects, which means that once you create them in memory, you can modify them, but for now, any time you make a change to such an object, well, you can't change the object. If you want to get a different version of the object, Python will create a new object in memory and you can reassign the variable to that new object.

So in this example, if I want to grab-- if I have the string car in memory like this and it's bound to variable s and I want to change the first letter to a b, I'm not allowed to. Python won't let me do something like, I want to change the letter at index 0 to a b. That's not allowed. You can get new versions of that particular string. So you can do some random expression to create the bar that you might want. But then the car remains in memory. So the car will still be there, we're just losing the binding from it.

So any questions so far on these strings? Mostly they're new data type. You haven't worked with them like you have with numbers, so it's a little bit different. Again, someone had a question. How do you get the A? Backward. Try it out in the console. I'm happy to answer questions, help you try it-- try it along with you, but that's what the console is there for. The shell. Here, that's what it's there for. It's just to try quick little things if you ever have a question, what if this or this and you get to try it out?

Now let's move on to some input/output. So far, the programs that we can write are pretty stagnant. There isn't much interesting things that we can do with them. There's no interaction with the user. So so far, when we've tried to output things, well, you might think, we have been outputting things.

So when we write in our console something like 3 plus 2, Python does show something in the shell. This is maybe how we interact with the user. But this is not actual true output. This is-- I call this peeking into the value of the expression. But if you were to write some expression like this in a file editor, Python wouldn't actually print it out. And so here's all the things that we've already tried today. We've created all these strings, we've got the length of s, we indexed. Anytime we typed these expressions in the shell, Python automatically gave us our value.

But if I were to type those exact expressions in a file editor on the left here, Python is not actually going to print these out. So this is the file editor. From now on, we're just going to work with files. I'm going to run it by hitting this little green Run button or hitting F5. Something happened in the shell. My program ran. It says here, it ran this file. But there's no output. Where was the length of s? Where were all these indices we've done before?

And that's because these aren't actual outputs. When we type them into the shell, that was just us doing quick little expressions in the shell giving us-- that's why I call it peeking into the value because it's not true output. If you want the user to see output and the shell is how we're going to show the user output from running a file, we have to explicitly tell Python, hey, I want you to show the output from this expression, or I want you to show the output from this command. And we do this using the print command.

So if we take our expression that we want to show the output from and wrap it in a print command, Python will then show that output and only that output. Can you imagine if we wrote a file that did all these operations and all these intermediary outputs were being shown? That would lead to a really messy file-- or a messy program. And so that's why we have a command where you can explicitly tell Python just the things you want to show to the user.

So here, if you want to print the length of s, we can wrap the length of in a print statement and then run the file. And now, the only thing that gets shown to the user is the thing I explicitly printed out, the 3. And then down here, if I want to print this other result-- the result of this other expression, I can wrap that around a print statement and Python will then print that one as well. But now I'm in charge of showing the user the things that I want to show them.

OK. So whenever you have a print statement, Python will print that resulting expression and then enter a new line. So as you saw here, we had two print statements one, around len and one around s at negative 3. And Python put the result of these expressions, each one on a different line.

Sometimes you might want to have expressions on the same line-- or the results of expressions all on the same line. So we can do that. We can put all of these different objects within the same print statement. We separate them by a comma within the print statement. That's down here. Python will print all of our objects no matter what type they are, and it will separate each object by a space.

So there's my object "the," there's my object the number 3; and there's my object the string "musketeers" and it printed it all on one line with a space in between them. And that's what this comma does, it automatically inserts the space. Now let's say you don't want a space for whatever reason.

What if we try concatenating these objects together? Remember, we saw concatenation, we said it doesn't automatically insert spaces, it just merges the strings together. And we run it? Well, I already gave it away. It's going to be an error, but let's see the error. It's a type error. It says can only concatenate strings, not integers to strings.

All right, makes sense. This is a string, this is not a string, so that's not OK, and this is a string. So instead of concatenating different objects together, we now have to remember to cast every object that's not a string to a string. So this line is exactly the same as the previous one except that b, which was the number the integer 3, is now being cast to a string. So I'm wrapping the b around the str. And that casts my integer to the string, and now Python is happy to concatenate these three strings for me. OK that's basically what I said.

So that's output using the print statement. Now how about input? We can get input from the user, not surprisingly, with a command called input. The format of input is usually like this. So we have the input command. In the parentheses, we give it a string. And then we usually want to save the input to a variable.

So the next few slides are going to go through step by step what happens when I have these two lines of code. Text equals input "Type anything," and then I'm going to print 5 times text. So when Python sees a line that says input and then some string, Python will automatically take the string within the input-- so in this particular case, here's my input command. The string inside the input is "Type anything," colon, space.

On the shell, Python will put that string for you. And then it will wait. It waits for the user to type some stuff in and hit Enter. As soon as the user hits Enter, whatever the user typed in-- so let's say the user typed in "howdy." Whatever the user types in, will be saved as a string replacing this input statement.

So you can think of the input like an expression. It's a weird one because it's waiting for the user to give us something. But in the end, the input gets replaced by the string version of whatever the user typed in. So the user can type in something-- numbers, letters, characters, anything. As soon as the user hits Enter, whatever the user typed in will be saved as a string replacing this input.

So in memory, the way this looks like is-- this is our memory cloud. Here is this-- here is this object that I've created, which is the exact characters the user typed in.

OK, well, if the user typed in "howdy," then what does this line end up? Being text is equal to the string "howdy." And that basically is What we've seen on the previous two slides when we've worked with strings. We're going to assign this variable and bind it to this particular string of characters.

Now the next line is easy. We're going to print whatever the result of repeating text is five times. So the print will show on the shell, howdyhowdyhowdyhowdyhowdy. Whatever the user typed in five times.

OK. Let's look at another example. In this particular one, we're going to ask the user for a number. And I want to print 5 times whatever the user types in. So num1 will, again, grab input. So what we're asking the user to do is to type in a number. So when the Python sees this, it prints "Type a number" and then waits for user input. Let's say the user types in the number 3. That gets saved as the string 3.

Again, so no matter what the user types in, it's being saved as a string. Even if it's a number, it's being saved as a string that number. So to Python, it's a character. To us, it's a number, but to Python, it's still a character. So num1 in memory basically becomes the string 3, just one single character 3. When I print 5 times num1, what is that going to look like? You guys tell me.

AUDIENCE: 33333.

ANA BELL: Exactly. Right. 33333. Because we're working with a string here, not an integer. If we want to work with an integer, we have to wrap our input statement with a cast statement. So again, this is what Python does. We can combine expressions together. In this particular case, we're going to combine the casting, the input, with the input statement. So now the user can type in for me 3 again. The input itself is going to be the string 3, but that line becomes num2 equals int, parentheses, string 3. And that-- I did it on the shell earlier today. When we cast a number-- a string to an int, it becomes the number, that int.

So num2 is then going to be 3. In memory, num2 is not the string 3 anymore because we've cast it to 3. So when we print 5 times 3, we're doing the mathematical operation 5 times 3, 15.

OK. Let's have you code. So I'm going to give you a couple of minutes. I'm going to have you write a program that is interactive. So it's going to ask the user for something and it's going to print something back to the user. So we're going to ask the user for a verb, and then I want you to print two things for me.

The first is whatever the verb that user typed in, you're going to write "I can whatever better than you" on one line. And then on the next line-- so with another print statement, I want you to print that verb five times. So if the user types in "run," you're going to write "I can run better than you." And then on the next line, run run run run run.

So the way these You Try Its work is I actually have some space here. I've already pre-written instructions for you. And all you have to do is fill in the code. So I'll give you a couple of minutes and then we'll write it together with suggestions from you and we'll see how far we can get, and we'll definitely-- we'll definitely finish it together so you don't have to finish it on your own. Yeah?

AUDIENCE: It this supposed to have [INAUDIBLE]?

ANA BELL: You should have this file. It's part of the zip file you downloaded for today.

AUDIENCE: How do I-- oh, here it is.

ANA BELL: Yeah. All right, does anyone have a start for me? So how can I ask the user for input? Yeah?

AUDIENCE: I think the question was [INAUDIBLE].

ANA BELL: Yep. That works for me. And I'm adding a little extra space here between the colon or whatever prompt you have just so that when the user types it in, it isn't right beside the colon or the end of this string. So as soon as we do this, the user will-- the program will wait, and the user will get to type something in. What's the next step? What's the first-- how can you how can you use this input?

AUDIENCE: Do print.

ANA BELL: Yep. Let's print something. "I can" in quotes. Yep, "I can."

AUDIENCE: Could you, like, outside of the quotes, put the question within [INAUDIBLE]?

ANA BELL: Yep. We can put a question. Yep, exactly. "I can," question, comma, because it's another object, and I'm happy to put a space in between it. "I can," question, another string, "better than you." Whoops. OK. There. And we don't need to write the full program right away. We can just test this little bit out. So choose a verb run. The one I gave you is fine. That looks good so far. All right. So then we can keep working on the second part. How can I print that verb five times? Yeah?

AUDIENCE: Print and then question times 5.

ANA BELL: Print question times 5. OK, let's run it and see what happens. Run. Not quite. I'm missing spaces, but this is an awesome start. How can I add the spaces in there? Yeah?

AUDIENCE: Parentheses, then verb plus, like, quotes with a space.

ANA BELL: In parentheses, yep. We can concatenate it with a space. Exactly. All of that times 5. Yeah, let's try that. Run. Yep. That looks pretty good. I do want to mention one thing. There is one improvement we can make to this program. If we look at the output here, the thing that we're actually printing out is this verb space. There's 1, 2, 3, 4-- and the last one actually prints it with a space at the end.

So a challenge for you, and the answer is a little bit lower-- I provide you guys with the answers to these, but a challenge for you is think about how you can change it-- change this last print statement so that this last run doesn't actually have that extra space. Just think about it. You don't have to do it right now.

OK. So with what we know so far, we can actually apply some of these ideas to a more numerical example. So Newton's method is a way to actually grab the roots of a polynomial numerically using this idea called successive approximation. We can't actually write the full algorithm with what we know so far, but we can write a really important part of it. The part is-- the part that we can write is the one that gets a next guess based on an initial guess.

So you don't need to understand how the algorithm works, but basically, the next guess based on an original guess looks like this. This is the formula. So the next guess is the original guess minus-- and we evaluate the formula for whatever polynomial we want to find at the original guess divided by the derivative of that function at the same guess.

So here's just some code we've got asking the user for input. What x do we want to find the cube root of? Then we ask the user for input, what guess do you want to start with? And then we can just print the current estimate cubed. So we just guess cubed. And then the next guess is just following the formula up here. The next guess is going to say it's my original guess. So the g that I read in from the student-- or from the input minus, and now I have a division.

The top of it is going to be f at g. And the computer is not evaluating f. We have to give-- we have to actually write down what the formula is-- the function is. We want to evaluate at g, so it's g cubed minus x. That's our function up there. Divided by the derivative.

And again, the program is not going to evaluate the derivative automatically. We're going to tell it what the derivative is manually. So the derivative of g cubed minus x is just 3g squared, so then we just hard code that in. And the next guest to try is just going to be that particular division and subtraction.

AUDIENCE: Is this a function of the derivative?

ANA BELL: I'm sorry?

AUDIENCE: Sorry. Is this a function for the derivative of [INAUDIBLE]?

ANA BELL: There are Python packages that allow you to do that, but for our purposes, we're just going to hard code it in this case, but yeah. So the way this looks in code is as follows. That's exactly what we had in there. And if we run this program, all it does is-- let's say we want to find the cube root of, say, 27, let's start with, I don't know, 5. It tells me that 5 cubed is 125, way too big, obviously. So the next guest to try is 3.69.

And that's all the program does. It doesn't take this next guess and do another guess. We haven't learned how to do such a thing yet, but we will in the next couple of lectures.

One other thing I want to mention is this thing called an f-string. It's something that became available I think a couple of years ago in Python with Python 3.6. It's a way more convenient way for us to print out mixtures of literal text and resulting expressions. So if you have a bunch of complicated expressions you want to print out, an f-string is the way to do it these days.

What we know is these first two lines. This is what we've learned in the past couple of slides. So if you wanted to have these two values and print this big number is whatever fraction percent out of the original number, if you actually run this in the Python file, you'll see that this comma here puts an extra space between my number and the percent. And that doesn't look very good. When you have 3%, you're expecting the percent sign to be right by the 3.

But this comma adds for me an extra space, so it looks a little bit weird. Which means that our solution was to cast things to strings. So if we wanted to have that percent sign be right beside the number, we'd concatenate this cast with the percent.

But f-strings allow us to do this all in one. So there's no concatenation to think about, there's no casting to think about. f-strings basically are this f and then a long string. And it's a mixture of expressions and things that I want to print literally to the screen.

So the thing that's not inside a curly bracket are all things I'm going to print literally to the screen. So the space is space, and then later on, percent, space, "of percent," those are all things that will literally be printed to the screen. Anything that's within a curly bracket is considered an expression in Python.

And so before Python prints out the thing to the screen, it's actually going to evaluate whatever num times fraction is, and it knows these are going to be variables. And then later on, fraction times 100, and then later on, none. These are all variables or expressions that it will evaluate before actually putting them on the screen.

And now notice, these expressions, we might have had to cast to strings beforehand if we wanted to concatenate them, but now we don't because they're in this special format with the curly brackets of the f-string. So just something to practice. I'll interchange-- I'll use sometimes this, I'll use sometimes casting, I'll use sometimes f-strings, but if you can use f-strings whenever you can, that's really the way to go in Python these days.

So the big idea, actually, even with f-strings, is that you can place expressions anywhere. We saw-- we placed expression-- I forget here-- where we indexed-- we placed an expression in the index. Now we're placing expressions inside print statements. And now we're placing expressions inside f-strings. So expressions can be placed really anywhere, which is pretty awesome. Very versatile. Python will just evaluate them and then just move on to the next lines.

OK. So the last topic-- I'm sorry. Any other questions about the inputs and outputs? Is that all right? OK. So the last thing that we'll talk about today-- and we will maybe talk a little bit about it next time, is conditions for branching. So right now, the kinds of programs we can write are basically very linear. We have a bunch of lines of code and they get evaluated one by one. There's no way to skip around, there's no way to repeat things, there's no decision points in the programs. The values that you get are just the values that are in the program.

Now we're going to look at ways that we can add decision points in our program. So if some value-- if some variable value is less than some other variable value, we want to evaluate some code. And otherwise, we'll do some other code. So some code can now be skipped in programs with this new-- with this new idea.

Before we go on to showing you exactly how to do that, I'm going to talk about another notion of equal in programming, and this might be more the notion of equal you might be used to in math. So the first notion of equal is the one we've already seen. It's assignment. It's done with one equal sign. The value on the right-hand side is bound to the variable on the left-hand side. That we've known.

Double equal in Python is how we tell Python that we'd like to know whether these two expressions are equal-- or equivalent. Sorry, not equal. So if we're going to be looking at equivalency, how do we express equivalency? Well, if something is equal to something else, we can say yes or no. We can say true or false. True or false should ring a bell. It's the Boolean data type that we saw last lecture.

And so now that we're going to show you equality or conditionals in programming, we're going to start talking about Booleans a little bit more. So expressions don't just have to be numerical. Expressions can actually give us Boolean results. So for example, an expression like 2 less than 3 is OK in Python, and this expression actually evaluates to a certain value. It's not a number, it evaluates to true, the Boolean value true, because, yes, 2 is less than 3.

The equal sign here, this notion with a double equal, is how we ask Python to tell us whether two things are equivalent. And this will be the Boolean value false. So here's a bunch of other operators that we can run on any type, really, in Python. Most of them-- most of the time we're going to run them on numbers, but they can be run on strings and things like that as well.

So obviously the double equal sign checks for equality. So if i is the same as j, this entire expression is replaced with true. And if they're not equivalent, this entire expression is replaced with false. If we want to check for inequality, we use not equal-- so exclamation mark equal is-- it means not equal. So if the number-- or whatever object i is not equal to object j, then this entire expression is true. If they are equal, then the entire expression is false. And then of course, we've got the less than, less than or equal to, greater, greater than, or equal to to work with as well.

We can apply these to strings. And with strings, it's important to be careful about case. So for example, lower case a equivalent to upper case A is false because they are not the same character.

Now that we're talking about Boolean operators, we can actually start to combine them together. So if I have the expression, for example, 2 less than 3, like I wrote on the board, that's true. But I can combine that expression with another one. Actually by itself, I can say, what is not 2 less than 3? And that will be false. It's the opposite of it.

But I can also combine Boolean expressions together. So I can say, what's 2 less than 3 and 3 less than 4? So 2 less than 3 is true. And 3 less than 4 is also true. So the combination of these two expressions-- of these two Boolean expressions is what is true and true? True. So if one is true and the other one is true, then both of them-- and both of them together are going to be true.

If one of these is false-- so is 3 greater than 4 is false, well, what's false and true is going to be false. So if one of these operators is false, then the entire expression is false. And you don't have to remember this truth table. You can always check it like I just did right here in the console.

But at a high level, when we're doing the "and" operator between two Boolean expressions, we need both of the expressions to be true for the result "and" to be true. The "or" is the other one we usually-- we can usually do. The "or" is always true except for when both of the operators are false. And it makes sense makes sense in English to write. If either operator is true, then the entire result is true. But when both are false, the "or" of both of them is false as well.

So here's a little example where we can use these operators in Python. So we can draw the little memory diagram as well. So pset_time is 15, there's my variable. sleep_time is 8, there's my other variable. I'm going to print sleep_time is greater than pset_time. So here, my print statement is going to grab that expression, which evaluates to false. 8 is less than 15 is false, so that's going to print false.

And then I have two more variables. These ones just happen to be Booleans. derive is true, drink is false, so drink and derive is going to be false because one of them is false. And so here, I've got this other variable both, and then I'm going to print false to the console.

OK. Quick You Try It for you guys. So let's have you write a program that saves a secret number in a variable. So that's going to be your program's secret. Presumably people using your program won't be looking at the program itself, they'll just be interacting with the program in the shell. So save a secret number in a variable, ask the user to guess a number, and then print either true or false. If it's the same as your secret or not.

So it's here in this You Try It down here. So you can start with something like secret equals and then put your favorite number there, 5 whatever, and then write the rest of the code. So ask the user to guess a number, print a Boolean depending on whether the guess equals this secret or not. So I'll give you a couple of minutes to write that. Yeah? Sorry.

AUDIENCE: Do you put the symbol "and" similar to [INAUDIBLE]?

ANA BELL: If you use a symbol "and," it's not the same. You have to actually type out A-N-D in Python. The "and" means something else. It's an operator with the bits of the number. So something-- it's not going to give the same answers. Yeah. Right, you're thinking about Java or C++ or something, right? Yeah. All right. Does anyone have a start for me for this program? How do I grab the user input? guess equals input?

AUDIENCE: [INAUDIBLE]

ANA BELL: Yeah. We can be nice. "Please guess--"

AUDIENCE: [INAUDIBLE]

ANA BELL: What's that?

AUDIENCE: [INAUDIBLE]

ANA BELL: We want the user to give us an integer-- yeah, a number. Exactly. So-- OK, yeah. If we leave it like that, then we're just grabbing the string. So we have to cast it to an integer. Exactly. Now what? How do I check for equivalency between my secret and the guess?

AUDIENCE: [INAUDIBLE]

ANA BELL: secret equal-equal guess. And you want to-- print this, yeah. Let's print that. OK. Run it. Let's guess something that's not the same. False. Run it again. Let's guess something that's the same, true. And we can guess something that's lower to just-- is everyone-- yeah?

AUDIENCE: Do you assign-- like for different [INAUDIBLE], did you assign the [INAUDIBLE] equal to guess to a container?

ANA BELL: Yep. Yep, exactly. Equal equals this thing. Yep, and then you can do whatever you want with that. Print equal or something. That's the same, but yeah, you can do other things with this variable. Yeah. Exactly. 5. Yeah.

If you want, at home, try to see what would have happened if you didn't cast it to an integer. See if the program would have crashed or not, or if it would have just worked but given a wrong answer. So why do we do Booleans? Booleans are important variables because they allow us to start thinking about writing programs that make decisions. The way we talk is we can say something like, if this is true, do this; otherwise, do this.

The Boolean part is if that something is true. So the something is true is going to be the Boolean that we can create in our programs, and then the do this is some sort of command, and then the otherwise do that is going to be some other set of commands.

So a really simple Boolean expression could be, it's midnight, you get a free food email, do you go get the free food or do you sleep? That's the very simplest decision point you can make. But with conditionals, you can actually write a pretty cool program that gets you to that free food.

So let's say this is a map of MIT, this is where you are, that's where the free food is. We can write a really simple algorithm using just conditionals that takes you to that free food. So the algorithm goes like this. So I'm going to say, I'm going to walk always in this direction. So I'm either going forward, backward, left, and right, I'm not turning. And I'm going to say the algorithm is always going to have my right hand be on a wall.

So if the right is clear-- so standing here, my right is clear, so I'm just going to keep swimming until I reach a wall. If my right is blocked but my forward is clear, I'm going to keep going like this all the way to the end of the room. If my right is blocked and my forward is blocked as if I would have reached the end of the room, I would have gone to the left. And if my right, forward, and left is blocked, if I'm over there, I would go backward. So I'd go backward.

So basically starting from here, I've made my way all the way around this room and I would go out the door down the hallway. And if the map of MIT doesn't have islands-- so if the free food isn't somewhere in an island in the middle here, if it's just a regular old maze, I would eventually make my way to the free food following this really simple conditional algorithm.

So how do we actually do conditionals in Python? How do we tell Python, hey, I want to create-- I want to insert a decision point right here? We do that using the keyword "if." And the "if" starts a decision block. Now the simplest decision block is just an if by itself. So if Python sees that if-- so there's some code that it's following, and then at some point, it reaches the if, the condition tells Python to check whether that condition is true.

If the condition is true-- so this is our decision point, then I'm going to deviate from my main program and potentially-- and do the code that's part of that condition. Those, I guess, two lines, dot-dot-dot inside there. If the condition is not false, I'm not going to go that route and I'll just keep following the rest of the main program.

How does Python know how many code lines to execute that's part of that condition? Well, it looks at the indentation. So notice here, I've put a few spaces, for these two dot-dot-dot code blocks here. Anything that's indented right after that if statement and that colon there is a set of commands that are part of that block. So anything here will get executed all at once. And that's a really simple if. Either you do the set of commands, extra commands if the condition is true, or you don't.

Now you can add an exception to that. So if the condition is true-- again, we're following the program, we reached this if conditional here. If the condition is true, again, we're going to deviate from the program and execute this other set of commands right here. Otherwise, the condition is not true and we're going to execute this other set of commands over here. So these guys over here.

So either we do this set of commands or the other set of commands, but we never do both and we never skip both of them. So either we do one set or the other. When we're done executing all the indented blocks, part of the condition or the other one that if the condition wasn't true, then we rejoin the rest of the program and continue executing. So this is all the rest of the program is at the same indentation level as our original if and else.

We can add a whole bunch of conditions. Not just an if, do this, otherwise do this. We can actually add a bunch of things to check using elif, which basically stands for else, if another condition, do this. So here's our program. We reach a decision point. If the condition is true like before, we'll execute this set of commands, but otherwise, the condition is not true, we're going to check another condition.

Else, if this other condition is true, we'll execute this other set of commands. Otherwise, here's another elif. We'll check another condition. If it's true, we'll execute some other set of commands, otherwise there can be more elifs. And at some point, we're going to rejoin the rest of the program.

Now these elifs are going to be-- each condition is checked one at a time. The very first one that's true is the one that gets executed. So we're never going to execute more than one because this is an if, else if, else if, else if. So even in English, you're only going to do one of these. You're never going to do all of them. It is possible to skip all of them, though, because if none of those conditions are true, you just don't do any of them. If more than one is true, you do the first one that is true.

If you want to have a catch-all, a version of the middle if, elif, elif, you just add an else at the end. So if none of those conditions are true, you can add an else, which says you just do this if nothing is true. I like what we had over here. If this one, otherwise do this. Well, if any of these conditions are true, do one of them, otherwise do this.

So here's an example. We've got pset_time, we'll just put some variables in there. sleep_time, we'll put some variables in there and run it, see what we get. I've got one code block here, an if, elif, and an else. So the first code block, the condition is, it checks that the sum of those two is greater than 24 and it does something. This is the block that's part of that condition. Notice, it's indented by usually four spaces.

Elif-- so if this one was not true, then I'm going to go ahead and check the next one. The next condition is that the addition is greater than or equal to 24, and then we're going to do this print statement here. And if neither of those are true, I'm going to do whatever is in this code block here. I'm going to do these two lines.

So this is my-- I call it a catch-all because none of those other conditions were true, so we're going to catch ourselves and do this-- do these commands here.

And otherwise, once we finish doing either this one or this one or catching whatever is left over in here, we're going to evaluate the print statement here and we're going to print "end of day" because this is the rest of my program. Notice, it's at the same indentation level as my original program.

So here is this-- oops, this program. So if pset_time and sleep_time is 22 and 8, the addition is more than 24. So this is going to enter this code block here and print impossible. If it's exactly equal to 24-- so 22 and 2, we're not going to enter this one, but we will enter this one right because it's exactly equal to, it's not greater than, so then we're going to print full schedule and then rejoin the rest of the program here and print "end of day."

And otherwise, if this is something low, less than 24 and not equal to 24-- so neither of these conditions are true, then we're going to enter the else. And we're going to evaluate-- or run these two lines of code here. So the two lines of code here are going to grab the absolute value of 24 minus the pset_time minus the sleep_time figuring out how much time we have left in the day. It's also going to print this line here and then rejoin the rest of the program to print "end of day."

OK. Quick check. Nothing to run-- nothing to write here, nothing to run. Think about this program. What is wrong with it? So I'm grabbing a number for x, a number for y, and then I'm checking if x is the same as y, I'm printing x is the same as y. So if I give it 5 and 5, I'm going to print 5 is the same as 5. And then I'd also like to print these are equal. What's the problem with this program? Yeah?

AUDIENCE: If x is not equal to y, it's still going to print these are equal because it's not [INAUDIBLE].

ANA BELL: Exactly. If x is not the same as y, we rejoin the rest of the program because the indentation level of this print statement is the same as the rest of our program. So how do we fix it? Indent. Yeah. We'll just indent that print statement in to be at the same level as the if statement.

So we can actually nest indentation statements-- we can nest conditionals because once we've created a conditional, it's just a code block. So here, I've got an if statement with its own code block. And inside that code block, I can actually have more if statements that are just going to be executed whenever this condition is true. So this is the inside code block.

So for example, the place where we would execute this inner code block is when x and y are equivalent because then I'm going to enter this code block here, this is true. I'm going to print x and y are equal. And then this second conditional here, y is not equal to 0, is also true. And then I'm going to print this one as well.

I've already done one of the conditionals. They're true, so I'm going to skip the elif, I'm going to skip the else, and I'm going to rejoin the rest of the program. All the other cases, when one value is different than the other, will either take me here in the else and then rejoin the rest of the program, or when they're equivalent, I'm going-- or here, I don't have-- I don't actually have a case for that one on the slides.

But when they're equivalent and y is equal to 0, I'm not actually going to enter this inner conditional because y-- while x and y were true-- were equivalent, which is true, y was equal to 0, so that not equal to 0 is false. It's just backward. And then we rejoin the rest of the program. Yeah?

AUDIENCE: What did you do to float [INAUDIBLE]?

ANA BELL: Oh. I'm casting the numbers to floats. I could cast them to ints as well. Yeah. Yeah. Just so I'm not comparing strings. Yeah. So now that I've introduced conditionals, it's important to do a little bit more practice to get a mental model-- a mental model of how to trace the code. And the visual structure of the code actually helps a lot.

And Python is unique in the sense. There's no other languages that actually force you to indent things. So other languages don't really force you to have this visual structure to match exactly what's going on. But it's actually really useful in Python. That's what I like about Python. It just helps you see things that are going on immediately, like this set of code is part of this code block. And so it helps you debug a little bit more efficiently.

But the more practice you get, the more you'll get used to tracing the code and knowing exactly if these variables have this value exactly where your code is going to go. So I'm going to skip this You Try It because it's just kind of tracing the code and I'm going to have you do this one-- or we can write it real quick. Or you can start and then we can write it together. It's a variation of the program you just wrote.

Instead of telling me whether the guess is true or-- is the same as the secret number, I just want you to print whether the guess is too low, too high, or the same as the secret number. So we're going to need to put a conditional in there, if some conditional, we're going to print something. So I'll give you about a minute and then we can write it together and then we can be done. Oh, yeah?

AUDIENCE: Can we have two if statements into the program?

ANA BELL: You can have two if statements in the program, yeah. And there's actually some exercises I have for you guys to try at home here where there are two if statements in the program, and just to see what happens. That starts two conditionals. So if some conditional, that one can be true, and if some other conditional, that one can also be true, and then both will be evaluated. It's not an else situation. Yeah. That's a good question.

So I'm just going to copy the input from before. Does anyone have a start to my condition? I just copied what we had before for the input. Yeah?

AUDIENCE: So if I have if x is [INAUDIBLE] and equal [INAUDIBLE]?

ANA BELL: Yep. Yep.

AUDIENCE: And then print--

ANA BELL: Print.

AUDIENCE: Your guess is "too high."

ANA BELL: Yep. "too high." good. Yep. That's a great start. So we can even run it and guess something that we know is too high. Perfect. Too high, yep. Next. Do you want to do an else or an elif? Yeah?

AUDIENCE: Actually, I would get rid of the equal sign because if we put in a 5, now it will still say too high.

ANA BELL: That's a good point. So if we run it now-- let's run it with a 5, it says too high. Exactly, yeah. So let's remove the equal sign. It's a good thing we debugged that. So we can do an elif, the guess is equivalent to the secret. And then we can print equal. Whoops.

Does everyone understand why we remove that equal sign from the greater than? Yeah? Because we would have missed, yeah. We would have mistakenly gone into that first path. But elif, we can have a case where the guess is equivalent to the secret, sure, and then we'll print equal.

And then the last one can either be an else because we know the only other option is guess is less than, or we can do another elif if we want to, but we can leave it as an else, and then we can print "too low." And then we can run it and we can guess all the variations. So something that's too high, something that's the same, and-- I'm not sure what I did there. I should restart my kernel. So we did something that's too high, something that is equivalent, and then we can do something that's too low. OK-- yeah?

AUDIENCE: What's the difference between having else, else-if [INAUDIBLE]?

ANA BELL: So there is no difference. We can do an elif guess is less than secret. That would-- the program would work just the same. The else is just quicker because we know there are no other options here.

AUDIENCE: Why wouldn't we just put a row of if statements?

ANA BELL: We could also-- in this particular case, we could also put a bunch of if statements in a row, but then we'd have to be careful that they are mutually exclusive. So like in the previous example, if we have a bunch of conditions that might all be true, all those ifs will execute. That's the thing-- because the if starts a block. The elif is just associated with that block. So either you do one or the other or the other. But if you have a whole bunch of ifs, then they might all be true and they'll all be executed. Yeah. Yeah?

AUDIENCE: Why don't we use parentheses with an else statement?

ANA BELL: Oh, we could use parentheses in the if-else statements. You mean like this? Yeah, we can do that, especially if we have a whole bunch of expressions together, but if there's just one, Python will automatically know to do the expression first and then do the if. Yeah. These are all wonderful questions, by the way. OK.

So as we saw, there was a little bug in our code. It's a good thing we ran it. I should have run it with a bunch of different options, but it's important to debug early and debug often just to make sure that you don't introduce a bug that will kind of carry on throughout the code. That's another big idea.

And then a quick summary of what we've learned. Input and outputs obviously make our programs interactive. We added branching as a way to introduce decision points in our program. And next time, we're going to do a little bit more branching and then introduce looping-- so ways to repeat commands in our programs. So I went a little bit over time. I won't do that again.