[SQUEAKING] [RUSTLING] [CLICKING]

ANA BELL: OK, let's get started with today's lecture. It's going to be more of a chill lecture than what we've done in the past, even though we've got quite a few things to cover, as you can see from this title slide. I'm not going to go super duper fast, so please feel free to ask lots of questions. And then the second half of the lecture will be really chill because we're going to be talking about testing and debugging strategies, so super high-level topic.

But first we're going to tie up some loose ends related to lists and relating to functions. So we're not going to introduce a lot of new syntax. These ideas are more optional in your day-to-day coding, but they're just really, really nice to know. So let's first start talking about this idea of a list comprehension.

So you've been writing functions that deal with lists. And one really common pattern that I hope you've seen so far is the following. So this code right here shows something that we've definitely coded together and you've definitely coded in the finger exercises and the quizzes. And so it is a really common pattern. So the idea here is you have a function that creates a new list where the elements of this new list are a function of the input list.

So the pattern here is we create a new empty list inside the function. We have a loop over every element in the input. And to each one of these elements in the input we apply the same function. So in this particular case, we're taking that element and squaring it. And each one of these elements we're appending to this new list, originally empty, until we've done this function to every element in L, and then we return this newly created list.

So since this is a really common thing that programmers do, Python allows you to do this exact functionality with one line of code. And the way we do this is using something called a list comprehension. So the way that we do a list comprehension, essentially taking these four lines of code from this function, we are going to write them in this one line of code that looks something like this.

So the idea here is with this one line of code, we're going to create a new list. We're going to have an iterator that goes through some sort of sequence of values, and we're going to apply the same function to every one of those elements. And the other optional piece that we can add inside this list comprehension is to only apply that function if some condition holds.

So let's look at this-- let's look at this example and see how we can convert these four lines of code to one line of list comprehension code. So we've got creating a new empty list. This is going to tell Python to create a new empty list for us. So just open and close square brackets. And within these open and closed square brackets, we're going to write a one-liner expression.

And this one liner is going to encapsulate these two lines of code here. So the expression-- sorry, the function we're going to apply to every element in L is going to be taking that element and squaring it. So on the right-hand side here in the list comprehension, we've got some e squared. Well, what is e? Well, it's going to be every element e in L.

So if we read this in English, we basically say Lnew is going to contain elements e squared for e in L. So it sounds weird, but it kind of makes sense, even if we read it in English. And behind the scenes, Python will take one by one each element in L, square it. And that's the sequence of elements it will populate this Lnew with.

Now, what if we add a condition to that? So let's say we want to create this new list of elements only for even elements. So we only want to square the even elements within my original list L. Well, if we were to write a function that does that, we have to add this extra condition here. So everything else is the same except for this if e%2 equal to 0. This tells Python to only grab elements that are even, divisible by 2.

So how do we write this in list comprehension form? So here's a new list. And this is the function to apply only if the test is true. In list comprehension, this is my new list. I've got the for loop is over here. And then the test to apply is at the end here, if e%2 equals 0. And then what is the function we're applying? It's just e squared like before. So the test just gets appended to the end of this list comprehension expression here. Yeah.

AUDIENCE: I see it running faster. Is there a reason to do that?

ANA BELL: Does it run faster? I'm not sure, actually. It might run marginally faster but probably not significantly. The reason to do this is because as you get more practice with it, this will be easier to read in code. And often, if you see a large chunk like this, your eyes will glaze over. You're not going to want to read a chunk like that. But if you see it all in one line, you're going to think, well, how bad can it be?

And so you can come up with really complicated list comprehension expressions. But usually, we reserve them fo really simple, really quick ways to create these lists that you just populate with some values right off the bat. So it just makes the code a lot easier to read.

So list comprehensions are pretty useful. If you get a little bit of practice with them, you'll find yourself kind of using them all over the place. And they basically replace code that looks like this. So these lines of code is a very generic way of writing this one-liner list comprehension.

So here I've got a function f that I would like to apply. This expr, expression, is the function I would like to apply to each element. This is the list I would like to apply that function to. And the test is going to be the conditional. In this particular case, this test means I apply it to every single element. But you can imagine having a function which, in the previous case, we would say lambda x%2 equals 0 as our condition.

And then the function that we're essentially replacing is this with list comprehensions. We create this new list. Again, this is the pattern that we saw in the previous slide. We loop through every element in the list. If that condition holds, append that function applied to each element. And then at the end, return the list. So this is just a very generic way to write a list comprehension.

So let's look at some concrete examples. So here, I'm not applying the function e squared to a particular set of elements from a list. I'm applying it to the sequence of values given by range. Remember, when we were talking about for loops iterating through things, they can iterate through integers following some pattern, like range 6, range 1 comma 9 comma 2, something like that. As long as you have a sequence of values you can iterate over, you can plop that into this list comprehension.

So you can iterate over lists. You could iterate over tuples. You could iterate over these direct ranges. You could iterate over a range of the length of a list. Whatever creates an iterable for you, you can put that in the list comprehension. So in this particular case, the way I read this is I've got something that I'm squaring. And what's the thing that I'm squaring? It's going to be each value in range 6.

So I think about it like, what is the sequence of values that I'm going to operate on? Well, it's going to be the numbers 0, 1, 2, 3, 4, 5. And the thing that I'm going to do to them is square each one of those values. So the end list that I get out of this one liner here is a list containing 0 squared, 1 squared, 2 squared, 3 squared, 4 squared, and 5 squared.

We can add a condition to that. So here I've got each element squared for e in range 8 only if e is even. So then the way I think about it is let's start off with what every element in the range is. Well, it's 0, 1, 2, 3, 4, 5 6, 7. The condition I'm applying to that is that it's even. So the numbers I'm going to end up with, I'm filtering all those to only contain 0, 2, 4, and 6 because we go up to but not including 8.

And then I'm going to square every one of those. So the end result from this list comprehension is a list containing the elements 0 squared, 2 squared, 4 squared, and 6 squared. And lastly, we've been doing just single integers in the resulting list. But as I mentioned, we can do more complicated things. So as long as we can write a little expression here for the thing that we'd like to calculate or add to the list, we can put it in the list comprehension.

So in this particular case, the element that I'm adding to my list comprehension or my resulting list from the list comprehension is a list itself. So each element in my resulting list is another list, right? And that inner list is going to contain two elements every time, the thing I'm actually iterating over and its square.

And I've got a condition here. So I've got the elements 0, 1, 2, and 3. That's the range. But I'm only grabbing the odd ones in this particular case. So the resulting set of numbers that I'm going to apply this to is going to be the number-- is the numbers 1 and 3 because those are the two odd numbers in range 4. And so the resulting list is going to contain two elements.

So this outer square bracket is the list that I've created. And its elements will be the element that I have actually iterated over and its square as a list, so 1 and 1 squared for e and e squared when e is 1 and then 3 and 9, 3 squared, when e is 3. Questions about that? OK.

So pretty cool. It's a really nice way to create lists really quickly. Like, if you wanted to create a list full of zeros, full of a hundred zeros, no need to do a loop. You basically do a list comprehension that says square brackets 0 for e in range 101-- or 100. And then you've got yourself a nice little list full of a hundred zeros.

All right, so think about this and then tell me what the answer is. So the idea here is we have this list comprehension. And just go through it step by step. It looks a little bit intimidating. But the first step is to look at for loop and ask yourself, what are the values I'm iterating over? Then look at the condition if there is one. There is one in this case. It's at the end here. So now what subsets of those original things you're iterating over are you actually keeping?

And then from those things that you're keeping, what function are you applying? It's the one right at the beginning. So think about it. And then I'll ask you to tell me.

So step one, what are the values I'm iterating over, the full values, not including the condition? Someone yell it out. Yeah, that list in the middle, awesome. OK, so xy, abcd, and then 7. And then what's the last thing? Is it the number 4.0 or a string?

AUDIENCE: String.

ANA BELL: Yeah, exactly, 4.0. OK, string, string. Step two, from this list, what are the values that I'm actually keeping based on the condition?

AUDIENCE: If they're a string.

ANA BELL: If they're a string. All right, which one's a string? Is xy? Yes. Is abcd? Yep. Is 7? Nope. Is 4.0? Yes, excellent. OK, good. OK, so then these are the elements that I'm keeping. And now what's the function I'm applying? And what's the result going to be? It's going to be a list containing--

AUDIENCE: 2, 4, 3.

ANA BELL: Yeah, 3--

AUDIENCE: 2, 4, 3.

ANA BELL: 2, 4, 3. 2 because that's length 2, 4 because that's length 4, and 3 because that's length 3. Great. And we've got ourselves a nice little list based on that condition, that sequence of values, and that function applied. Yeah.

AUDIENCE: Why does it return a list?

ANA BELL: Why does it return a list? The whole thing?

AUDIENCE: Or I guess I thought it would return 2, 4, 3 on separate lines.

ANA BELL: Oh, yeah, so we're not printing things out here. When we're writing this as a list comprehension, we're essentially telling Python to create this resulting list of values. That's just what a list comprehension does. And so this expression here with these outer square brackets around our entire expression tells Python that the resulting thing is a list. Yeah, this is a good question. Other questions? OK, so that-- oh, yeah, question.

AUDIENCE: Does it support multiple conditions?

ANA BELL: Does it support multiple conditions? Yes. So at the end here, you would say "if," and then you could wrap them in parentheses. I don't know if you have to, but just to be safe, I would wrap my conditions in parentheses. And you'd use "and" or "or" or whatever you want to combine the expressions-- or the conditions with. Another question? Yeah

AUDIENCE: [INAUDIBLE]

ANA BELL: This one, the lambda? Here, this is a lambda function that we talked about I forget when, a couple of lectures ago. It's basically an anonymous function. And all it does is return True all the time. So the test will always be True, which means that when we do if test parentheses e, this will always be True in this particular case. But when given a different lambda function, that might not be the case.

OK, so let's move on to the next topic. The next, I guess, two topics we'll be dealing with functions. And I want to wrap up a couple of things here just to give you a couple more ideas regarding functions. So the first one is actually related to this last question is the idea of a default parameter. So this is going to be a way for us to add parameters to our functions that get some default value. And that's what that lambda thing actually was in that example. But hopefully, this piece of the lecture makes that a little bit more clear.

And then the second part regarding functions we're going to go over is the idea of functions as objects, kind of working up on that. And we're going to see what happens when we return a function object from another function. We've seen functions as parameters to other functions, but we're going to see what happens when you make a function be the return value of another function. But that's in a little bit. For now, let's look at default parameters.

OK, we've seen this code before, triggering flashbacks. So this is bisection_root. I'll go over it just to remind ourselves what it does. We've got this code inside this function we wrote a long, long time ago. And then we decided to wrap it in a function so that it's a really nicely useful piece of code that we can run many, many times.

So the parameter to this function was x, a value we'd like to approximate the square root of. And the code we're using to approximate is using the bisection search algorithm, which initializes some variables, namely epsilon, how close we want to be to the final answer; low and high endpoints, we remember that; and then an initial guess, the halfway between low and high.

And then we keep making guesses between low and high-- being the midpoint of low and high, as long as we're not close enough to the final-- we're not close enough to the final answer. So we're going to either reinitialize our low endpoint or our high endpoint depending on whether that guess was too low or too high. And then within the loop, we make another guess using those changed values of either low or high based on if or else.

And then we keep doing this process of making more guesses at the halfway point as long as we're still farther than epsilon away. OK, that was a recap of what we've done so far. The interesting thing that we had done with this function was-- or when we turned it into a function was to return our approximation.

So this guess, instead of just printing it to the user, we returned it so that it could be useful in other parts of the code. And so when we called the function, we just said name of function and then some value of x.

Now, there are situations where a user would want to change the value of epsilon. Right now, the way we wrote this code, epsilon is set to 0.01. And whenever you run the function, it always finds the approximation to the square root of x to that precision, 0.01. Now, sometimes, depending on the application, the user might want an even better approximation, so 0.000001, or they might not care to be as precise, and they want maybe approximated to 1 or to 5 or something much bigger than 0.01.

So what are the options in this particular case, for these scenarios? One option would be obviously to go inside our function and say, well, I'm going to change epsilon to be something super duper precise, 0.000001. And so people who call this function will always get an approximation to that precision. But what about people who don't want it that precise?

So all the function calls are going to be affected by making that change, and so that's not really desirable. We'd like to let the person who makes the function call be in charge of what precision they'd like. Another option is to put epsilon outside the function, so to say, OK, well, the only parameter is going to be x. And let's not set epsilon within the function. Let's let the user maybe set epsilon outside the function. And then they can use-- and then our code will basically pop up one level to the global scope and use the epsilon that the user chose.

Not a good idea because as soon as we allow somebody using our code to make their own variables within our code, we're putting our trust in somebody else's hands. And they might forget to reset epsilon. Or they might forget to set it to begin with. And so just using global variables is not a good idea in the first place. We'd like to keep control of the epsilon that's being used inside our function.

So unsurprisingly, the last option is going to be our best option. Let's just add epsilon as another parameter to the function. So there it is. We've got bisection_root again as a function. We've got a parameter x. And we have epsilon as a second parameter that the user can call the function with.

So other than that, the function body is exactly the same, except that right now, when we make a function call, we have to pass in epsilon as the second parameter. So in terms of code, this is the bisection_root with epsilon as a parameter. And so now the user can find the approximation to 123 to 0.1. It's 11.088 in case you were wondering. And then the approximation to 123 to .000001, which is 11.0905.

So, much better, the user can now be in charge of deciding how close they'd like the approximation to be for every one of their values. But notice that this code is kind of verbose. And really, most of the time, maybe the users don't want to be in charge of setting the epsilon. Maybe they don't know what a good epsilon might be.

So how do they know that they should choose 0.01 by default? Maybe that's something you could put in the function specification for anyone using your function. But you're going to rely on users reading your specification, and that's a little bit scary. So instead, the functionality that would really like to have is to say, OK, I want to write a function that does take in two parameters. But by default, one of those parameters is something that I set, as the person who's writing this function.

So what I would really like to have is epsilon to have some sort of a default value. So if users don't know what to call it with, the code will just use that default value. And otherwise, if the user is more experienced and they know they'd like an epsilon of 1 times 10 to the negative 10 or whatever it might be, then they can be in charge of setting it. So most of the time, we want to call the bisection_root function without an epsilon parameter so that it may use a default one. But sometimes we'd like to allow the user to actually set the epsilon.

And so to that end, we're introducing the idea of keyword parameters, also known as default parameters. And they are set like this. So the bisection_root function definition still takes in the thing we'd like to approximate the square root of, x. But the second parameter here, epsilon, will be equal to something inside the function definition.

So we as the people who are writing this function are going to say the default value of epsilon is 0.01. So that means when we call the function down here, if the user makes a function call without explicitly passing in a second parameter, Python will use the default one that the person who wrote the function set. So Python will run bisection_root of 123 with epsilon being 0.01.

And otherwise, if the user does want to override that epsilon, they can just pass it in themselves, and that default value of 0.01 will be overridden to be 0.5. And so in our code here, this is the bisection_root with the default values. And so you can see here if I run it with 123, even though there are two parameters here for the bisection square root function, Python doesn't complain because it's using epsilon as 0.01. So I run it, and it runs just fine.

But in the second line here, if I actually want to use 0.5 as my default-- as my epsilon value, it overrides my default parameter, and it calculates the square root of 123 with epsilon being 0.5. Questions so far?

So now that we've introduced default parameters, there's a few rules about making function calls. When you create the function definition, so over here, when you're the one defining a function and you decide to allow some default parameters in your parameter list, everything that's a default parameter needs to go at the end. You can't switch these around. You can't say epsilon equals 0.01 comma x. Python will not allow that.

So any time you have default parameters, they always have to go to the end. That's the only rule for making the function call or making-- defining the function with default parameters. And then once you have default parameters, you can actually call the function in many, many, many different ways. And I know some of these will be confusing. You might not know whether they're allowed or not. You can never go wrong with the last one, as we're going to see you in a bit.

So the first one here showcases what happens when you give values for everything that's not a default parameter, in this case just x. If you just give a value for nondefault parameters, Python sets default parameters for everything else. So not a big deal. Alternatively, you can pass in, just like we have in the past when we write our own functions with multiple parameters-- you can pass in one at a time in the same order values for every one of those parameters, default or not. And if you pass in values for all of them, Python will not be confused, and it'll just match them one at a time.

Variations on that, you can always pass in a value for a parameter name. So looking at the function definition, we can see the parameter names, the formal parameters are named x and epsilon. So when you make your function calls, you can actually explicitly tell Python something like this, x equals 123, epsilon equals 0.1. And if you have more parameters, you say that parameter equals whatever value you want to run it with.

And so that will not confuse Python. And if you do it in that way, you can actually do it in any order you'd like because Python will just assign each one of these variables to be whatever you told them to. So worst case, you just do something like this, where one at a time you just say what the formal parameter is and its value, and then Python will not get confused.

The ones at the bottom, though, is where we run into trouble. So for example, if you put the default parameter first and then you put an actual parameter-- sorry, you put the default parameter first and then any parameter that's not a default one afterward, Python gives an error because the default ones have to go after the nondefault ones.

And the last one doesn't actually give an error, but Python, remember, matches parameters one by one. So it's actually going to find an approximation to the square root of 0.001 to an epsilon of 123 because it's just mapping the parameters one at a time. And so that's not an error, but it's not exactly what we want it to do. Questions about this?

OK. So now let's move on to another thing, another sort of nuance about functions. And we're going to go back to the idea of functions being objects in Python. So I drew this picture back when we first learned of objects-- of functions as objects. So I'll just do it again just to jog your memory.

So remember that when we make a function definition, inside the memory Python creates an object. As soon as we see just this function definition, Python doesn't care what code is inside here. This code does not run. It only runs when it's being called. And right here, I have not made a function call at all.

All Python knows at this point is that there is a function object inside memory, and its name-- its name is is_even. And this is exactly the same as creating an integer object inside memory and giving it the name r through a line like this or creating a float object in memory and giving it the name pi. It's just some object with some name.

And so that means that we can have some code that looks like this, which is going to essentially create an alias for that function object in memory. So here, the name is_even refers to that function object. And I'm telling Python that I would like to refer to that function object using the name myfunc as well. So both myfunc and is_even are names that point to this object in memory.

It's not a function call. I'm not trying to figure out if some number is even. I am literally giving another name to this function, this code that does this thing here. And so that means that if I have two names that point to the same object, if I am going to invoke those two names, as I do here, with some parameters, Python is going to say, well, I'm going to run the code pointed to by these names with these parameters.

So they will both run the code that they're pointing to. This is even, and so it's just going to return True or False. We've seen this before. So remember, just another name for that object in memory. So we've seen already how we can pass functions as parameters to other functions. And now we're going to see what happens when we return a function from another function.

So we're not returning a function call here. We are returning a function object. So in this particular code, we have only one function. It's named make_prod, and it happens to have some stuff going on inside it. So what's the stuff that this function will do? Well, this function itself will create another function. So this g only exists whenever make_prod exists.

The main program, you can think of it as this level of the code in terms of indentation-- the main program does not know about g. G is only defined inside make_prod. So when we first run this program as is, there's no function call being done. So the main program does not know anything about the internals of make_prod. So make_prod creates its own function here.

And then all it does is return this function object. Notice it's not a function call. There's no open close parentheses with a parameter in it. It's just the name g. It's this function object. That's the key thing here. So let's run two codes, this one and this one. They will do the exact same thing. They're going to call make_prod with some parameters.

And then we're going to see what happens when we return this g. And notice already it's looking slightly different than what we've been doing before. Yes, we have a call to make_prod here, but we've kind of chained another function call right after make_prod. We've got make_prod parentheses 2, parentheses 3. And so this is kind of like-- I think of it as chaining a bunch of function calls together.

And this is possible, as we're going to see when we step through the function environments that are being created-- this is made possible because make_prod, this function call, returns a function itself. So let's step through the code on the left very carefully. And then I'll step through the code on the right, which will do the exact same thing. And hopefully it will clear up confusions if we do it twice.

So this is the code from the left. Let's say we have this exact program here. I've got one function definition. And then I've got one function call here. And then I'm going to print the return value. So as soon as I run my code, Python creates my global environment. And in the global environment, this is the scope of the main program. What do we have?

Well, we have one function definition which has some code within it. I don't care what it is at this point because I don't have a function call. So then the next thing that I need to do is go down here and say val equals-- so I'm going to create a variable val in my global environment. And I'm going to make a function call.

So function calls are done left to right, just like expressions. And the first thing Python sees is this function call, make_prod parentheses 2. It's a function call, so we need to create another orange box because a new environment gets created every time we make a function call. So here I have my scope, my environment for make_prod.

And I'm currently just stuck here, trying to figure out what this is going to return, just the red box here. Well, every time I have a function call, I need to look at the function definition. And the function definition says, well, there's one formal parameter "a" that I need to map to the actual parameter. So the thing I'm calling make_prod with is 2. That should be pretty straightforward, right?

And then I can move on to do the body of make_prod. OK, so the body of make_prod says, I would like to create a function definition. The name of this function is g. So there is g. And it contains some code. Again, I don't care what this code is because I'm not making a function call to g yet. Right now, I'm just defining g.

So far, so good. So this g, I want you to notice, only exists inside this call to make_prod. The global environment does not know about g at this point because we only define g inside make_prod. It's here, right? I didn't define it outside of make_prod, so the global scope doesn't know about it. But make_prod does know about it.

And so the only way that the global environment can know about g is if this make_prod function somehow returns g. So if we pass g back as a parameter-- as a value, sorry, to the main program scope, the main program can know about g. But otherwise, g is kind of stuck in this little subtask, little environment of make_prod. And the main program doesn't know about it.

And so that's what this code is doing. It's essentially saying, well, I've made my definition, and now I return g. So here, this g and its code-- and the associated code, so this object pointed to by g, is going to be returned back to the main program. So now the main program knows about this object g that has some code associated with it, this line here where it returns a*b.

So the thing that I've boxed in red down here is the return value from make_prod 2. And make_prod 2 returned g. So this, you can essentially say, is g.

Is that OK? Does that make sense? We're passing functions along, not function calls. And so this is just a function named g. And so now, this line of code, val equals, if we replace the red box with g, val equals g parentheses 3.

So g parentheses 3 is another function call, right? Just clearly, we look at it, it's a function call. It's got a function name, parentheses, and a parameter. And so since it's a function call, we create another scope for this function call. As before, we look at what g takes in as a parameter. It's a variable named b, a formal parameter b. And we map it to 3 because that's our function call, g parentheses 3.

And then we have to do the body of g. The body of g says return "a" multiplied by b. Well, I know what b is. It's 3 because you just called me with that value. But what is "a"? The scope of g has no "a" within it. So thinking back to our function, our lecture on functions, if a function call doesn't know about a variable name within its environment, within its scope, it moves up the function call hierarchy.

So it says, who called me? Where was g defined? Well, g was defined inside make_prod. And so it was called from make_prod. Does make_prod have a variable named "a"? It does, right? And its value was 2. So we didn't need to go any further up the hierarchy. We've already found a variable named "a," so Python will use b is 3 and "a" is two. Multiplies that to be 6.

And then the g function call can return 6. It returns it back to the main program because that's where this function call was being done. Remember we had this replaced with g parentheses 3 out in this global scope here. And so that 6 gets returned back to the main program. And then val becomes 6. And we print 6.

OK, so that was showing you how to chain function calls together. And this was only made possible because make_prod as a function returned another function object. If make_prod returned, I don't know, a tuple or an integer or something that was not a function, this code would fail because the return from make_prod would be--let's say it returned the number 10. The return from make_prod would be replaced with 10, and then Python would see this line as 10 parentheses 3, and what the heck is that?

And so it would completely fail. And so this is only made possible by the fact that this make_prod function returns a function object. And so we're able to chain these function calls together.

So let's look at the exact same code, except this time instead of chaining them in a row, let's explicitly save the intermediate steps. So what I'm going to do is say make_prod parentheses 2 I'm going to save as a variable and then make that variable call the 3, the second part of my chain from the previous slide.

And it's going to do the exact same thing. So here I've got the global scope just like before. I've got a function definition for make_prod. So this is the name make_prod. It points to some code. And then I've got this variable doubler that's going to equal something. So it's a function call. The function call says here's my environment for make_prod with its scope. So in this particular scope, I've got my formal parameter "a" that maps to 2.

And then the function body itself creates this variable g. That's just some code, exactly the same as before. Any questions so far based on what happened in the last sort of example and here? Or is this OK so far?

OK. So now I've set up my code. And this is where the interesting part comes in. Make_prod is going to finish its call by saying, I'm going to return something. And the thing it returns is g. So it returns this name g. It happens to be a function object, but think of it as anything else. We're basically saying doubler equals 10 or doubler equals some list or some tuple.

Doubler is going to be some value. This value is just code associated with a function. So in my main program scope, I've got doubler equals g, which based on the memory diagram we did 5 or 10 slides ago, it's like when we had myfunc equals is_even. I basically have two names for the same function object. Doubler is a name, and g is the other name. And they both point to this function object.

Does that make sense? Is that OK? OK. So now that I've got two names that point to the same function object, we can just use this doubler in the next line. And this doubler is like saying g parentheses 3, except that I'm using the name [AUDIO OUT]