[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. OK? 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 reached-- 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 closed 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 and square it. And that's the sequence of elements it will populate this Lnew with.

OK. 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==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==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: Is 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? [LAUGHS] And so you can come up with really complicated list comprehension expressions. But usually, we reserve them for 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.

OK. So list comprehensions are pretty useful. If you get a little bit of practice with them, you'll find yourself 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 x%2==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, 9, 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 this 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. And that inner list is going to contain two elements every time-- the thing I'm actually iterating over and it's 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 the 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.

AUDIENCE: That list in the middle.

ANA BELL: 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: A 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 string. All right. Which one's a string? Is xy?

AUDIENCE: Yes.

ANA BELL: Yes. Is abcd?

AUDIENCE: Yes.

ANA BELL: Yep. Is 7?

AUDIENCE: No.

ANA BELL: Nope. Is 4.0?

AUDIENCE: Yes.

ANA BELL: 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 and 4.

ANA BELL: Yep. 3--

AUDIENCE: 2, 4.

ANA BELL: 2, 4, 3. 2 because that's length 2, 4 because that's length 4, and 3 because that's length 3. 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?

AUDIENCE: Yeah.

ANA BELL: The whole thing?

AUDIENCE: Or I guess I thought it would just 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 just kind of 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.

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. Is there a question? Yeah?

AUDIENCE: Isn't the lambda [INAUDIBLE]?

ANA BELL: This one, the lambda?

AUDIENCE: Yeah.

ANA BELL: Here, this is a lambda function that we talked about, I forget when. A couple 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(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 0.5 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.

OK. 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 223 to 0.000001, 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 we'd 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 overwritten 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 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, 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 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 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 non-default 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 non-default 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 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 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 my_func as well.

So both my_func 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.

OK. 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(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 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. 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 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. 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 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(3). So g(3) is another function call. 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(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 within it.

So thinking back to 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. 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 2.

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(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(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(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 this is 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. Just 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 five or 10 slides ago, it's like when we had my_func 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? That's 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(3), except that I'm using the name doubler, which I saved it as on the previous line.

So g(3) is another function call. Create another environment for g or doubler or whatever name. And here, I've got one formal parameter b. Its value is 3. And then we do the same trick where you ask, what is the value of a? I'm going to look up the hierarchy of things that got called to see what is the first value of a that I grab.

And the first value of a that we grab is the 2. And so we're going to multiply the 2 with the 3. And that 6 gets returned back to whoever called it, which was out here in the main program scope. And so this val will be equal to 6. And that's it.

Questions? Which one was easier to understand, this one or the one where we did the chaining? Just show of hands. Who liked this one more? Who liked the chaining more? Oh, interesting. OK. Was the chaining just easier to grasp because there were less names? OK, cool. I'm glad I showed it first then. Any questions though? Yeah.

AUDIENCE: Is there a particular reason we do it this way compared to the chaining?

ANA BELL: No reason. In fact, you would want to do the chaining way because then you avoid extra lines of code. And again, with practice, it just becomes really easy to know what's going on. Yeah. OK.

So that might have been confusing. Why do we bother doing that? Because that particular example, all we were doing is multiplying two-- or I guess doubling a number. We could have easily written that code to double a number without actually returning a function. That seemed way overkill for what that code was trying to do.

Well, it was showing you what you can do with an easy example, and you would definitely never ever write functions returning other functions for such simple examples. But it's really a method for cases where you have larger pieces of code that you'd like to write because if you're trying-- so if you're writing a larger piece of code, some software project, and every single function you'd ever want to use is kind of defined at the top level in the main program, it would become really messy.

And so there are cases where you would like some functions to only be visible or accessible by other functions. And so you'd only define those functions within the scope of other functions. That's one thing. The other thing is using this sort of chaining method allows you to have some control over the flow of control of a program. And so you can imagine in the example here where you basically create this-- you have this line here. And at some point you return g.

And you don't want to do the doubling right away. So you don't want to do val equals doubler right away. You can imagine having a bunch more lines of code here that do other stuff before you actually do the doubling. And so, in that case, in this larger, more complex program, you're essentially interrupting the flow of control here. You're not doing the doubling right away, but you did grab this function back. And then you can maybe do other things with that function before finally doing the doubling.

And so in that case, you can basically execute some code partially, do some other operations, and then finish executing at the end after you've done these operations. So again, for this example, it doesn't make much sense, but in a larger piece of code, this idea of functions returning functions is just another tool to achieve these ideas of decomposition abstraction, which leads you to write more organized code, more robust code, more easy to read code, and so on and so on. So you don't have to do this, but you do have to understand what it means for a function to return another function.

Any other questions? OK. So now we're going to do the last piece of today's lecture, ideas of testing and debugging. This lecture is usually pretty dry, so I'm going to try to make it more fun, as fun as I can. The reason why we introduced this lecture now is because I'm hoping that by this point in the course, you've had a chance to do some testing and debugging strategies on your own by kind of a trial and error thing on quizzes and on P sets.

So you've gotten a chance to maybe use the Python Tutor, you've gotten a chance to use print statements, various things like that, and see what works best for you, what doesn't work at all, things like that. So you've maybe gotten a little bit burned by some of these strategies. But I hope that by you being burned by some things that you've tried that worked, that didn't work, you'll maybe appreciate this lecture a little bit more than if I just showed you this lecture back on day one or day two or something like that, because it's a lot of common sense stuff, but there's a little bit of actual strategy as well in this particular set of slides.

So your programming experience so far, I know this is certainly mine, is I hope that when I run my code, it immediately runs perfectly. But instead, what ends up happening for me, is I run my code, and it immediately crashes. I've got my red errors on the side, and I get a little bit flustered.

So this is exactly what happens, probably for you too. And the idea here is that you want to write the code in such a way that it makes it easy to test and debug. And I know I always say this, and I actually don't always practice it, but it's important to write the code by writing it by adding comments as you're writing the code. So writing specifications, writing comments for yourself as you're actually writing the code, not when you've finished it is, very important. It helps you as you're writing it, or when you're coming back to it in a couple of days.

Modularizing the programs also helps. So if you see a chunk of code that you're copying and pasting all over the place, you'll want to plop it out in a little function that you call multiple places. So ideas like that kind of employ this defensive programming mechanism, and it allows you to perform really easy testing and validation when it comes time to do that, and then possibly debugging when it comes time to do that.

So the lecture is going to be divided into two pieces. The first, we're going to talk about testing and validation. Some nice testing strategies. And then we're going to talk about some strategies for debugging as well. So the testing and validation part is where you come up with a set of input test cases and expected outputs. And all you're doing is running the test, running your code to make sure that the expected output matches the output that you actually get from running the code.

The debugging part is where one of your tests don't match the expected output. One of the outputs that you get don't match the expected output. And at that point, you have to figure out why the code is not working, obviously. So before you even test your code, as I mentioned before, you have to set yourself up to do the testing and debugging.

So to ease this part, it's important to write documentation very well. So when you're writing your own function, not functions that we've given you, document the docstring. What are the inputs you expect? What should the function do? What should the function return? Things like that.

If you're writing the code in sort of a strange way or if you use some piece from Stack Overflow or something like that, document it to make sure that if you're looking at it a week from now you still remember what that piece of code did. So really, really simple things like that can make a really big difference when it comes time to test and debug.

Breaking up the code, obviously, into smaller chunks is very important because if you're copying and pasting the same piece of code over and over again, you remember to make a change in one place, you might forget to make that same changes in all these different places. And so that'll be very frustrating when it comes time to actually run and debug the code.

So once you have code that's written, you would start the testing process. You remove all the errors. Static semantic errors and syntax errors are really easy to remove. Python immediately tells you, index error on this line or syntax error on this line. Those are really easy to figure out.

Using a paper and pen or typing it out in your Python file, you come up with a bunch of test cases. And for each one of those test cases, the way we write on your micro quiz test cases, you say what you expect the output to be. So when you actually run it, you don't need to remember what this output should be. It's just written down somewhere on paper or on the screen.

So when you're creating a bunch of test cases, you can create some different classes of tests. So hopefully we're modularizing our programs, which means that we're creating functions. The simplest classes of tests are called unit tests, and these tests basically test a function with different inputs. So what you're going to do is come up with a bunch of different test cases for one particular function and run these test cases on the function.

If they all work, perfect. But if they don't, or if you find a bug as you're writing test cases in the code, you'll want to perform regression testing. And regression testing means that as you find a bug, you add a new test case for them, or as you fix a bug, you run the code-- you run the same code with all of the previous test cases to make sure that the bug you fixed didn't introduce errors in a previous test case.

So there's a bunch of iterations of unit testing and regression testing to test all of these different modules, all the functions in your program. And at some point, you're ready to do integration testing. And in integration testing, you've got all these modules, for example, as you did in Hangman. You've got all these little functions that do individual things.

You put them all together into a larger program. In Hangman, it was a big while loop where you check all these different things that the user might input, and then you call the different functions you wrote. And as you find errors in the integration, when you've written code that integrated all these different pieces together, you might have to go back and do more unit tests for some of the functions that you wrote.

OK. So you've done unit testing, regression testing, and integration testing. What are some actual testing approaches? How do you actually create these test cases to run your code? So I guess the most natural way to write a test case is just intuition about the problem. So given a docstring, what are going to be some natural boundaries, some natural values of x and y for which you test this code with.

You guys tell me. What's some values that we could test this code with? Think about the boundaries to the question.

AUDIENCE: That means the difference x to y is [INAUDIBLE].

ANA BELL: Yeah. 3 and 4 is good. So x is less than y is a good one. Vice versa. 4 and 3 is another one where y is less than x. We could test them being equal. What about 0 and 0? What about 1,000 and 1,000? So we could do extremes, we could do bigger than, less than, we could do equal, things that.

So mathematical functions are kind of easy to apply this idea to because they just have natural boundaries. But often, there are functions which don't have these natural boundaries, and then we might be stuck doing random testing. And in random testing, obviously, the more test cases you have, the better chance you have of finding a bug.

But there are actual techniques for coming up with test cases. So the first one is called black box testing. Second is called glass box testing. Now, in black box testing, you're going to treat the code of the function as a black box. So we don't even look at what the code is doing. All we're looking at to guide writing our test cases is the specification, the docstring.

And so hopefully the person who wrote this function wrote a really nice docstring because that's what we're going to use to write our test cases. So the way that we're going to write a test case for this square root function is by saying, what is the value of x and epsilon according to these constraints here? So obviously, we're not going to test the code with values that don't match those constraints because the person who wrote this function doesn't guarantee that this function will work out of those weird values.

So the good thing about black box testing is if we're the ones testing this function, we're only using the specification to write the test cases. So if, for example, this person implemented square root using approximation method, I don't care. My test cases will work if the person changes their implementation to use the bisection method. My set of test cases will still work even if the person who wrote this function changed the black box, the implementation. So black box testing is really nice in that respect.

And so for this particular function, here's a bunch of test cases that I would run it with. So obviously, x being 0, perfect square less than 1 are kind of nice ones to test, irrational values, and then a bunch of extremes is also good to test. And then epsilon, the same. We've got some reasonable values of epsilon, and then some extremes. And we can even mix and match. We can have 0 and extremes epsilons, and perfect squares and extremes epsilons, and things like that. So lots more test cases than this, but this is a really good start.

In glass box testing we're going to use the code itself to guide the test cases that we write. So if we write something, a test suite that's path complete, that means that we're going to hit every single path inside the program. So that means we have to look at the code to guide the test cases that we're writing, which means that we're going to have to write a test case for the code hitting the if part of a branch, we have to write a test case for the code hitting the else part of a branch, or the L if part of the branch.

If we have a for loop, we need to write a test case where the code doesn't go through the loop at all, it goes through once, or it goes through many times through the loop. Same with while loops. We write a test case so that the code doesn't go through the while loop at all, it matches the condition once, or it matches the condition many times. So you can imagine that this glass box testing leads to a whole lot more test cases, especially when we have a whole bunch of different combinations of all of these conditionals and loops and things that we'd like to hit.

The problem with glass box testing and having a path complete test suite is that we might accidentally miss a bug. So here's an example of a code that's not correct. So it finds absolute value of x. If x is less than negative 1, we return negative x, else we return x. So a path complete test suite could be testing 2 and negative 2.

The 2 brings us through the else. So we return 2. And the negative 2 brings us through the if. So we return 2. We might say this code works, but it doesn't. We already can tell that negative 1 is returned incorrectly as negative 1. And so in addition to testing all the paths through the code, we'll also want to look at boundary condition, especially for conditionals, when we do glass box testing.

OK. So we have a whole bunch of test cases. We've run our code with all these test cases. And then, at some point, we've gotten an output from a test case that does not match what we expected to do. Then we have to do the debugging process. And this is where a little creativity is required.

There is no recipe, like there was in glass box testing and black box testing, for writing test cases. There is no similar sort of recipe for debugging a program. There is a lot of experience that's needed, a lot of times that you've seen a bug crop up in order to figure out what the problem might be. And so a lot of experience writing code is very useful in the debugging process.

There are tools to help you do the debugging process, but there aren't many tools to actually do the debugging. You kind of just have to do it. So there's tools built into Anaconda. They're not very good. I've used them. Python Tutor, obviously, is a really good one, especially for small programs, because you get to just go step by step and see the values of each variable as the code is running. So I like that a lot.

Print statements are also really good, but you have to know where to put them. And you have to use them effectively. So in that sense, if you're not as familiar with print statements, Python Tutor might be better suited for debugging. But no matter what, it's important to be systematic. Don't just start changing random variables or random conditions, and then run the code through the tester again. That's not going to work very well for us.

When we see error messages in the debugging process, these are really easy to figure out. IndexError. Oh, shoot. I got to check my indices. Maybe I went over. If you see an IndexError, you should probably print out the variable that you're indexing into or indexing with.

TypeErrors. Oh, man, look. I'm casting a list to an integer. What is that going to do? Nothing. It's going to give us an error. Or here, I'm dividing a string by an integer. Again, something really simple to fix. NameErrors, of course. Here I have a variable that I've never initialized. And then SyntaxErrors basically mean things like your indentation is off or you're missing a parentheses or something like that.

Logic errors are a lot harder. These ones, you cannot just look at the line and say, this is where the problem is. These ones happen when your output does not match the expected output. And this is where engaging another part of your brain is very important.

I've definitely done this a lot. I've had some errors, I went for a walk, come back, and I figured it out, or I figured it out in the shower, or I figured it out in bed. So thinking a little bit before you even start the problem is good for these logic errors. Drawing pictures, taking a break, talking to friends, all these are really good.

Explaining the code to something else, somebody else, is also a really nice thing to do. That's me explaining the code for something we're going to do in a couple of minutes to my son. He's seven now, and he's doing Scratch. So that's pretty cool. But he was helping me debug, and now I'm helping him debug. Yeah.

Or you can explain code to some inanimate object like a rubber ducky. Now, having said that, you guys came on a good day because you will all get to have your own rubber duck. Different kinds. Grab your personality duck that matches your personality after class.

I've got *Minecraft* ducks, giraffe ducks, princess ducks, police ducks, elephant ducks. Whatever ducks you'd like come grab one. Use it for your quizzes. Use it for your P sets. Whatever you'd like to use it for, go for it. OK. So hopefully it comes in handy. Use it well.

All right. So we're not quite done yet, though. OK. So I will give you a little bit of debugging tips though. So I know I said it's a creative process. I said it's really hard to come up with a recipe to do the actual debugging. But there is maybe one really nice way to do it.

So the idea behind debugging is to basically use the scientific method. Like I said, don't just randomly change things expecting for it to work out. What you want to do is look at a bunch of test cases that failed. It's possible that they all have something in common. And that might lead you to a small piece of code in your program that is the one that you should be focusing on changing. So you want to look at the data, form a hypothesis, and try to see if another test case also fails that particular one.

As you're doing the debugging method, if you really have no idea about where to start, try putting print statements at reasonable places in the code. So when you first enter functions, when you first enter a loop, write all the values of the loop variable and all the variables that you're creating in the loop or modifying in the loop and things like that. And if all else fails, using the bisection method is a really nice way to try to solve the problem.

So bisection method and debugging basically says, put a print statement about halfway in the code. If everything looks right for all the variables at that point, you know the problem is after this. If something is wrong, you know the problem is in the first half of the code.

Then put a print statement in a quarter of the code. And then, at that point, see if all the values at that point match what you expect them to be. If they do, great. You know the problem is in the second quarter, I guess. Yes, the second quarter. And if they don't, the problem is in the first quarter. So the bisection method is a nice way to try to debug the code.

So what we're going to do in the last bit of lecture is we're going to debug some code together. That's in the Python file. And then what I have included in today's zip file is actually a Wordle game that I wrote. It's like 12_wordle.py or whatever. And it's buggy. So I introduced some bugs in it. And if you'd like to practice debugging, you can try to fix the Wordle game to get it to work. And then you can play it yourself or you can amaze your friends and get them to play your game in case you'd like to do something like that.

So before we end, I would like to actually do some debugging with you, just to show you the bisection method for debugging. So the code we're going to debug is this one right here. And I've already included the fixed code step by step, but we're going to talk through it together. So this function is buggy.

It's a function called is_pal that takes in a list x. And it's supposed to return true if the list elements are a palindrome and false otherwise. So using the input abcba cast as a list-- so the input is going to be the string a, string b, string c, string b, string a-- this list is a palindrome because it's the same forwards as it is backwards.

So if I run it, it should print true. OK. So that test case worked well. But now, what about the second test case? Surprise, it's not going to work. If I pass in the list ab, so my input is going to be the string a and the string b, this is not a palindrome. So I expect it to print false, but it prints true.

So I have a nice test case here that I can make fixes with and see whether it actually gets fixed. Now, of course, abcdefghijklm, this also doesn't work. So this is another test case that's not going to work. But I don't want to use this long one as my test case. I want to use the simplest test case I can find that doesn't work. So ab seems like a really nice one to test with.

OK. So now, the first thing we want to do, now that we've figured out the input I'd like to test with, is put a print statement about halfway through the code. Yes, there's only like five lines of code here. So there's only probably one place that makes sense to put a print statement. But let's just work with me here.

So the print statement could be put right here, right before the if statement. So I've got two lines of code that do something, and then an if. So let's just put it right before the if. Scroll down. Step two. Here I go. I've put my print statement right before the if.

Now we can run the code again. So I'm not going to run the one that worked. Let me try to run the one that didn't work to figure out what the problem is. So I run this. The print statement is printing the temp, so the reverse of x, and x. So what I'm expecting-- and I should have probably written this over here, what I'm expecting to get. What I'm expecting is to see the reverse of ab, so ba, and then the original x, ab.

But I don't. So I see ab and ab. This first one should be ba. So already I have something that's unexpected, and so I know the problem is going to be in these first two lines of code, somewhere in here. All right.

So then what I would like to do is figure out which one of these lines of code is the problem. So I'm going to put another print statement a quarter of the way through the code. OK. Well, there's only one more place to put it. So let's put it in here.

I've got another print statement right before the reverse. So what I'm going to be checking is, before the reverse, the value of my temp variable and my original variable, and after the reverse, the value of my reverse variable and the original variable. So what I'm expecting to see is this one here, they should be the same, ab, ab. But this one here, I'm expecting to see ba, ab.

So run it with this buggy example. So before the reverse, I'm expecting ab and ab. And I do get that, so that's good. I'm happy to see that. And then after the reverse, that's my problem. I'm expecting this one to be reversed, but it's not. So now I know the problem lies here, temp.reverse, because here, in this print statement, here temp and x were as expected. So what do you think the fix should be to the reverse? Yeah.

AUDIENCE: We need to add parentheses.

ANA BELL: Yeah, exactly. We need to add parentheses. This is a function. We need to call it like a function. So let's do that fix. We've done it here. So here, I've added the parentheses to the reverse. And run it again.

So now what I'm expecting is before the reverse, I need to see ab, ab. So this one should be the same. It shouldn't change because I didn't do anything to that temp equals x. And after the reverse, I'm expecting the temp to be ba and the x to be ab, unchanged.

All right. Let's run it. So before the reverse, everything looks OK. temp and x are the same. After the reverse, look at that. I've got my ba as my reversed variable. I'm happy. But then my x has also changed. I'm sad. Yes.

AUDIENCE: You have to make a copy of x.

ANA BELL: Exactly. There's a clue. We see a clue. We've made a change to temp and x has also changed. So as was suggested from the back, we need to make a copy of x. What we've done here is called, when I did temp equals x, on a mutable variable. What did I make?

AUDIENCE: Alias.

ANA BELL: An alias. Exactly. So let's make a copy of that x right here. So hopefully that fixes things because I've run out of lines to fix. So if we run this code again with ab and see the output before the reverse, temp and x should be the same. And they are. They're both ab, ab. And after the reverse, the temp should be the reversed ba, and it is. And the x should remain the same, ab. And it's false, so it's not a palindrome.

Last thing I need to do is double-check my original test case, the one that actually worked before I made all my changes, to see whether it still works. And it does. So that particular list is a palindrome. So that still returns true.

So that's it. So I've got a couple-- or just one, I guess, list comprehension for you to try on your own to write. And then, of course, the buggy Wordle game for you to try as well.