[SQUEAKING]

[RUSTLING]

[CLICKING]

ANA BELL: All right, so let's get started on today's lecture. So today we're going to be doing one of two lectures on the topic of recursion. And you may or may not have heard of recursion. It's a programming technique and a way to algorithmically solve problems. It's not something that's going to come easy because it's going to force our brain to think about problems that we've seen in a completely different way.

So you don't have to use recursion if you don't want to, but there will be problems where the idea of recursion and applying-- or writing recursive code is going to come a lot more naturally than writing code that we have been so far. But I'm just warning you, it's going to take a little bit of forgetting everything we've learned about loops and things like that to train our brain to think recursively for the next two lectures.

To help you, we will have an interactive portion of today's lecture. So think about whether you want to come up on stage, or whatever this is, the front, and be a part of the interaction. You'll be forever immortalized on the OpenCourseWare-- awesome, I love it-- on the OpenCourseWare videos.

All right, so let's think about iterative algorithms that we've seen so far. So iterative algorithm basically means we are writing code that has a loop within it, so either a for loop or a while loop. Writing code with these for loops or while loops lead to iterative algorithms, so things that do some task for some repetition.

So the idea of iterative algorithm is that there are some variables that capture the state of the computation. So each time through the loop, these variables will change their value, essentially capturing what the values are at each step in the loop. So when we're writing these iterative algorithms, we basically think about, what is something that's changing each time through the loop? Like we keep a running sum, that's the easiest example.

What is a variable that's changing each time through the loop, kind of like a counter that keeps track of how many times we've been through a loop? When do you stop? So for for loops, you stop after you've exhausted a sequence. For while loops, you stop when you have a condition that becomes false. And then at the end of the loop, you have some sort of result that you've been storing and accumulating or changing each time through the loop. So that's an iterative algorithm. And we've been working with these a lot.

So to show you-- we're going to go through the next few slides showing you an iterative algorithm to do multiplication. It's going to be very, very simple. But we're also then after going to look at the same problem, which is doing multiplication, but in the context of recursion. And hopefully that gives you a sense for how we think about the exact same problem we're trying to solve, multiplying two numbers together, in a completely different way.

So this is not the function that I want to write with iteration. I don't want to create a function named mult and then return a*b. I don't want to use the built-in function. I want to assume that I don't know how to do a star, a multiplication. And so instead, what I'm going to do is I'm going to rely on-- let's say I know how to do addition. I'm going to rely on the idea of addition to actually write my multiplication function.

So let's think about how to make multiplication iterative. We can have a loop that adds a to itself b times. That is the definition of multiplication. So let's write a function that does this using a for loop. Then we'll write it using a while loop. With a for loop, we're going to write this iterative algorithm.

It's capturing the state of the computation, just like we said we should. So the for loop will iterate-- will have my range of values, being from 0 to b. So we're going to repeat this loop b times. The variable total is capturing my state of the computation. It's keeping track of what the total is at each step through my loop. At the end of the loop, I return the total. So very, very simple iterative function here.

Now, let's think about another iterative solution. Instead of keeping a loop variable b that goes from 0 all the way up to b-- or what was in my loop variable? N, I think. Yeah, instead of keeping a loop variable n that goes from 0 to b, let's work our way backward. And this time, let's use a while loop just for fun. Let's say that I'm going to start at b and count down to 0.

So again, going and repeating some task b times. So what I'm going to do is I'm going to have some counter that starts at b and decreases down to 0. Again, within my loop, I have to keep track of the result. So my total in the previous code is now being called result in this code.

And so what I'm going to do is my iteration will start right at 0. And then I'm going to keep adding a to itself b times. So the code looks like this. I've got my while loop this time instead of a for loop. I'm going to start out with knowing what b is. And I'm going to decrease b by one each time through the loop. So here I've got b equals b minus 1. So that's capturing the state of the counter at each iteration.

The result, just like the total in the previous slide, is capturing the state of my sum at each time through the iteration. And at the end, I return result. So hopefully very simple, very review code here. But now, let's look at the code in a recursive sense. So here, let's not look at the code yet. But let's think about is there some thing that we're repeating over and over and over again?

If we recognize it, we can think recursively. OK, so let's try to figure out this recursive pattern. So I work best with examples, like actual numbers. So instead of using an abstract a and b, let's use a is 5 and b is 4 as an example. So let's say I want to use the star operator-- that's basically the function I'm trying to implement-- the star operator between 5 and 4.

So in the iterative sense, we said that's 5 plus 5 plus 5 plus 5, adding 5 four times. The idea of recursion is that we're trying to take our original problem, which is using the star operator between two numbers, and try to solve a very similar problem, if not the same, but in a slightly changed way.

So instead of saying I'm going to multiply 5 by 4, what I will do is recognize that 5 times 4, which is my original problem, can be rewritten by extracting out one of my 5's. So I'm going to take a 5 out and add it to 5 times 3. So this is my recursive pattern. I'm using the star operator between 5 and some number.

But if I extract a 5 out, I can use the star operator between 5 and a slightly smaller number, 1 less than 4. Well, what if I do-- what about 5 times 3? Can I do the same thing again? I can, right? For 5 times 3, I can again notice that I can extract the 5 out again. And I have 5 plus 5 times 2. And then I can do the same thing again to figure out what 5 times 2 is, I can again extract a 5 out and be left with 5 times 1 or 5*1.

And so notice the thing inside the box is basically me solving my original problem, which is using the star operator between 5 and some number. But that number is changing each time on each line. At some point, I can say this problem is so easy that I know the answer. So 5*1, so a number multiplied with one, is just the number itself. And so at that point, I can say I don't need to continue dividing my problem into smaller and smaller pieces.

So just to bring the point home, let's use parentheses to illustrate which pieces I'm replacing where. So I've got my original problem, applying the star operator in the multiplication on 5 and 4. I extract the 5 out, and I recognize that I can have 5 plus and then solving 5*3.

I need to have some trust here, right? I don't know what 5*3 is. But if I decompose that problem in the exact same way, I can extract the 5 out of that and add it to 5*2. So the thing in the boxes are equivalent. And then, again, 5*2, I'm going to recognize this is the same problem I've been trying to solve. Let's apply the same solution, which is to extract a 5 out and add it to the multiplication of 5 star one less. So again, the two boxes are equivalent.

So this idea here, where we're recognizing the same problem and kind of dividing it, dividing it, dividing it, having this trust that at some point we're going to divide it so much that we've reached a fundamental fact that we can solve is this divide step. So we're going to divide it all the way up to-- all the way down here, where I've got 5 plus 5*1. At this point, I can say, well, 5*1 is going to be 5. This is a basic fact that I can just solve.

I don't need to divide this problem any further. So once I solve this fact, I can start building back up my answer. And I can start passing the answer back up the chain of multiplication calls. So if I'm at this step here, and I figured out that this is 5*1-- this 5*1 is equal to 5, I can just replace it with a 5. And then I can build up the solution to this 5*2 because now 5*2 is just five plus 5.

So this is going to be 10. It's just simple addition. There's no more multiplication, which is the thing that we were trying to avoid. So then the 5*2 gets replaced with 10. And I'm still building back up my solution until I get to the 5*4. So I was trying to figure out what 5*3 is, but before I could do that, I had to solve the rest of the two lines beneath it. But now I can finally solve it. It's just 5 plus 10. That's the similar problem I was trying to solve.

So I can replace the 5*3 with 15. And finally, my original problem was trying to figure out 5*4, and now I can finally solve it because I've finally built back up my solution as 5 plus 15. Any questions about these steps? It should be pretty straightforward. I know it's a weird roundabout way of figuring out what the answer is. But what I'm trying to get at is trying to recognize the problem that we're trying to solve and then solving a very similar problem just slightly changed.

In this case, we're multiplying 5 star one less than what we were just trying to figure out. So in terms of the recursion for this particular problem, multiplying a with b, we recognize that a*b is going to be a plus a plus a plus a b times. If we extract an a out, just like when we extracted the 5 out and added it to something else, we'll recognize that that's just a plus a plus a plus a plus a plus a b minus 1 times.

OK, well, that a plus a plus a b minus 1 times is just our original problem, just multiplying a with b minus 1. So this is my recursive step. We recognize that a*b is equal to a plus a*b minus 1. So I'm using the same operation I'm trying to find here down here in my quote, unquote "solution."

But this is not the end of recursion, because if I just had this as my definition, then I would have infinite recursion. I don't have a way to stop. And so this recursive step, in conjunction with a base case, something that we know fundamentally about the star operator, is going to give us our solution. So we knew on the previous slide when we multiply a with 1, we just get back a. So our base case, very simple case of multiplication between a and b, is going to be when b is 1. And that's going to be a times b is equal to a.

So these might look like the mathematical definitions that you might come up with in a math class. And we have them right here. So if b is not equal to 1, a times b is a plus a times b minus 1. And then the base case is when b is equal to 1, a times b is equal to a. So with these two lines, we can actually come up with the code, the programming version of this function.

So here we're creating a function named mult_recur. And its parameters are going to be a and b. So I'm multiplying a with b. And I have to encode these two cases, when b is 1 and otherwise. So we usually start with the base case. It's the simplest to think about. So when b is 1, a times b is equal to a. So if b is equal to 1, then what is a times b? It's just a, right? So the function can just immediately return a.

So that's our base case. Else, this is going to be our recursive step. We're not going to return a, but we will return this, a plus a*b minus 1. Well, the a is just a plus and this, the star operator between a and b minus 1, is just the function again. Isn't that really cool? We're using the function name in the body of this function that we're defining.

And it's not a problem because the parameter to the one at the bottom in the recursive step is changing. I'm not calling mult_recur with a, b again. That would be very silly of me because that would lead to infinite recursion. I'm not making any progress towards a base case. But I am calling it with b minus 1. So this function will just keep calling mult_recur with a, with b, b minus 1, b minus 2, b minus 3, and so on, until it gets to b is 1. And then it'll build back up the solution, just like we had in the slides with all the parentheses that we were replacing.

OK, so let's step through the Python Tutor, and I will show you how it actually looks when we make all these function calls. And then we'll do another example. So here I've got mult_recur with a is-- I think I ran it with 5 and 4, just like the one we've been looking at. So this is going to be my main function. It makes a function call to mult_recur-- [COUGHS] oh, excuse me-- 5, 4. So my a is 5, and my b is 4.

This little blue thing here is one function environment. Like when I draw boxes on my slides that are orange, they do them in blue. OK, now, in this function call, what do we do? Is b 1? No. So we go in the else case, and we return 5 plus-- what happens next? Does anyone know? Yeah.

AUDIENCE: Mult_recur is going to run again with a as 5 and b as 3.

ANA BELL: Yes, mult_recur will run again with a as 5 and b as 3, exactly. It is a function call, right? So as a function call, we are going to create a new environment. So here's, boom, another box. My previous box is currently hung up. It cannot finish, because it's trying to figure out what 5 plus the result of this function call is. But this one is not done yet. It's still figuring out its result.

So we've put that one on hold. And now we're trying to figure out mult_recur 5, 3. Well, what is mult_recur 5, 3? It's going to be b is not 1. So this one will also go in its else. And it will recur-- it will return 5 plus what? Exactly, the function call when b becomes 2, exactly. But notice it is another function call, right? So here I have, boom, another box.

Now I've got two function calls, this original one back here, which was waiting on this one that I've highlighted here. But now this one that I've highlighted here had to make another function call down here. So I've got currently three function calls in the works that are trying to figure out what their results are.

All right, finally, this mult_recur 5, 2 is going to make another function call. So its b is not one. So we're going to go into the else. And what is its else going to say? We're going to return 5 plus-- and it's another function call.

So now I'm four function calls deep, and I haven't done any sort of visible work. I've just kept kind of kicking the can down the road to try to figure out what the values are. And everybody's waiting for somebody else to finally return a value. So this first one is waiting for the one right underneath it to return a value. But this one is waiting for the one underneath it to return a value. And this one is waiting for the one underneath it to return a value. That's the chain of calls.

What's this last one going to do? Is it going to make another function environment? No, it's going to return a, which is 5. OK, there's my return value 5. So this one will return the 5 to whoever called it. And whoever called it was this one here, mult_recur 5, 2. 5, 2 was trying to figure out what 5 plus this bottom function call was.

Well, now it can figure out that it's going to be 5 plus 5. So its return will be 10. This one returns a value up one level to whoever called it. And that was mult_recur 5, 3. And now mult_recur 5, 3 can finish its job. It was trying to figure out what 5 plus its function call was, which we figured out is 10. So this one can figure out it's 15. And finally, this last value can return back up to the original function call, 5, 4. And 5, 4 will return 5 plus the 15 that got returned, which is 20.

OK, questions about what just happened? Does everything make sense? All right. So let's look at one more example. I mean, we'll look at a few more examples this lecture, but let's look at a real world example for now. This one will hopefully illustrate the difference between iterative algorithms and recursive algorithms in a more real-life setting.

So let's assume that in this real-world setting, a student asks for a regrade for an exam. In an iterative setting, we have basically one function call, regrade, or whatever you want to call it. There's my student. How is the student going to iteratively get the regrade? Well, the student will be in charge of basically looping through each staff member.

So the student goes to the instructor and says, can I have a regrade please? The instructor may have graded one problem. Maybe they didn't, but they will regrade the problem that maybe they were in charge of. Then the student will go to the next person on staff, the TA. Can I have a regrade, please? Let's say the TA maybe regrades the problem they were in charge of. Maybe they didn't, but in any case, they'll give the score back, or they'll answer the student's request.

The student then goes to the next person on staff, the lab assistant. Can I have a regrade, please? The lab assistant might regrade the problem they're in charge of, whatever, gives the grade back to the student. The student is keeping track of all these regrade scores that they're getting to figure out what their new total score is. And finally, the student might go to the grader who did probably most of the work, asks for the regrade. The grader will dutifully agree to do the regrade and pass back the values.

So here, notice the student is in charge of iteratively going to every single person on staff and getting the result back. And the student is keeping track of what their new score is. Obviously, the staff members will too for the purposes of assigning grades, but the student is as well. So the student's basically adding up all these values. But there's only one function call. So I've denoted the function call using just this black circle here.

OK, so that's iteration. We know how to do that. We've been doing this for a really long time in this class. But now let's look at the same problem recursively. So in recursion, we've got these two steps. There's the problem of decreasing our original problem into smaller problems until we reach some sort of base case. And then at that point, we have the task of building back up our answer.

So in the recursive setting, again, I've got my one function called to regrade on behalf of the student. But the student will only interact with one person, maybe the instructor. The student will not interact with anybody else in staff. The student will just go up to the instructor and say, hey, I would like a regrade for this exam.

Now, the student is going to wait. The instructor is also a function call to regrade. So maybe the instructor didn't do any of the grading, but the instructor will make their own function call to the TA. Can you please regrade this exam? The TA maybe graded one problem. They'll keep track of the problem they need to grade, but there are other problems that need to be graded. So the TA will then make their own function call to the lab assistant. Maybe the lab assistant graded some problems. And then the lab assistant will also further the request, sort of passing along the function call to the grader.

So we have the task of doing the grading as a function being passed along all of the staff members. When we reach the base case, which is the grader that probably knows-- probably graded the last question, we've got the answer being passed back up the chain of function calls. So the grader will say, all right, I've graded my problem. There's nobody else I need to ask. So here's my score.

So this score is being passed back up the chain of function calls to the lab assistant. The lab assistant will take that score and add it to their score, passes it back up the chain of function calls to the teaching assistant. The teaching assistant adds that score to their score. Maybe they graded a problem. Maybe they didn't. But anyway, they're compiling the results, little by little, back up, until it passes it to the instructor. And then the instructor says, here you go, this is your score.

So you see the difference, right? The student is the iteration. They ask everybody on staff. So they interact with everybody on staff. But in recursion, the student is basically hung up waiting for an answer until we've gone down all these chain of function calls and the answer has been built back up. So the student is not keeping track of the answer at all. They only get the final answer at the end. Did that help at all? OK, I've refined this example a couple of times. Hopefully this is good.

So the big idea in recursion here is I've got these quote, unquote "earlier" function calls, the ones I've made way back at the beginning. These function calls are just waiting on results to come back. They're not doing any useful work at the beginning. They only do useful work when they're assembling the results after getting a return back from later function calls.

So hopefully that gives you a sense of how we can apply recursion. Now, what exactly is recursion? So algorithmically, it's a way for us to come up with some solutions to some problems in this divide-and-conquer approach or decrease-and-conquer approach. You have your original problem-- you divide it so much into the same problem just slightly changed, until you reach a base case. That base case can kick off the conquer step and start passing back a value that you can start assembling from your earlier function calls.

Now, semantically, as we saw in the example where we multiplied the functions, we've got a function that calls itself. Obviously, it's not calling itself with the exact same parameters because that would lead to infinite recursion, and that's not what we want. We're going to call ourselves with a slight change in our parameters, in such a way that we will reach our base case. And once we reach the base case, then, again, we kick off the conquer step, and we can start reassembling back. And you saw how the function calls do that when they help each other back up.

OK, I'm going to give you a couple of minutes to try this. So complete the function that calculates n to the power of p for these variables. So if you come up with the mathematical definition, it will be a pretty straight translation to code. I did include two base cases here. So maybe a base case is when n is 0 and another base case is when n is 1. Figure out what you should return and then how to write this recursive step.

So I've got a line. 50ish is where you can type in the code. All right, what's my first base case? Yeah.

AUDIENCE: If p is 0, then it'll return 1.

ANA BELL: Yep, if p is equal to 0, then we can return 1. Oops, just one time. What's another base case? P is 1; we can return 1. Awesome. How about my recursive step?

AUDIENCE: We can return n times n.

ANA BELL: Yep, we can return n times--

AUDIENCE: [INAUDIBLE]

ANA BELL: Like this? Now, let's assume I don't know how to do **, how do you rewrite this in terms of the thing we're trying to write? There was a solution back there.

AUDIENCE: Power_recur n, p minus 1.

ANA BELL: Yep, we can do that too. Yep, exactly. So here we're assuming that we don't know the ** operator, otherwise this would be a very easy function to write. We are trying to define the ** operator ourselves using this function named power_recur. So we're just going to call it again down here with n and p minus 1. So if we run it, this will give us 8. Yeah.

AUDIENCE: What's the necessity of having the p equals 1 [INAUDIBLE]?

ANA BELL: Yes, great question. What is the necessity of this? There is no necessity. I actually just included it there to just show you how we can have two base cases. So in this particular case, we would actually never hit this one, if n is greater than 1, because we always stop here. If the user gives us 0, we would just return that one. But it would work if we completely removed that as well. Yeah, great question.

OK, let's look at one more example. And this one is the one that I'm going to ask for some participation. I would like four of you to come down with me. But before we do that, let's think about factorial. So the definition of n factorial is n times n minus 1 times n minus 2 times n minus 3 down to 1. What is a base case? What is the simplest thing that we know the factorial of? You guys tell me.

AUDIENCE: 0.

ANA BELL: What is n-- what is 0 factorial? 1, good. I chose 1, but both could work. If n is equal to 0, you can also return 1. Or we can do n is equal to 1, return 1. What's our recursive step? Do you recognize the recursive pattern here? N factorial equals--

N times n minus 1 factorial. If we extract the first n out, n minus 1 times n minus 2 times n minus 3 and so on is just n minus 1 factorial. And so our recursive step just says it's n times the same function factorial with n minus 1. Is everyone OK with that? Cool.

OK, so let's look through this example with some participation, so four people. One, yes. And you'll be on OCW forever, you guys. Two, yep. Two more. Yes, thank you. Thank you. Awesome. OK, and I'll have you guys stand right here. I'll ask you guys to come in one at a time as we are working through this exam.

So we're just going to demonstrate sort of once again what happens when we make function calls. Do you want to just stand right here behind my computer? Thank you. Yep, behind my computer. Cool. OK, perfect. OK, so I'll just stand here. So I am going to be the main program. You run this code, I am going to be the main program.

I am going to keep track of the variables and everything that's in this global scope. OK, so in the global scope, just like we have been in the past, I've got a definition for the my factorial function here. And this is just some code. At this point, I've just defined it. I don't care what it actually is. But I have one function call. So my one and only job is to print the result of factorial 4. I have a pretty easy job.

So what happens-- you guys, the audience, tell me what happens when I've got factorial 4. What is this? Do I just know right off the bat what factorial 4 is? No. It is a function call, right? So as a function call, what do I need to do?

AUDIENCE: Create an environment.

ANA BELL: Exactly. I need to create an environment. OK, so you'll be my first environment. Hello, my name is-- you can put it on you. There you go. Hello, my name is-- and then you can step right over there. So you are my first function call. Your name is fact for factorial. Awesome. So I have just called you. What is your job? So you guys tell me, what factorial 4's job is from running the code. Are they going to do the if or the else?

AUDIENCE: Else.

ANA BELL: The else. So this is your job. You keep track of that. Your n is going to be 4. And your job is to return 4 times factorial of 3. Do you know what factorial of 3 is right now? No. So what do you need to do?

AUDIENCE: Call someone.

ANA BELL: Yes, please call somebody else. Who are you going to call? Next. What is your name going to be?

AUDIENCE: Factorial of 3.

ANA BELL: Your name is also factorial, exactly. And you are going to be called with n is equal to 3. So you can stand right beside factorial of 4. Very nice. So now, notice, we've got two function calls. Both of their names are factorial, right? But they are completely separate function calls. They are completely different environments. They have their own n values. They have their own jobs to do.

Just because their name is factorial for both of them does not mean that they'll interfere with each other's variables. Very, very important point to make. Factorial of 3, do you know what factorial of 2 is?

AUDIENCE: I do not.

ANA BELL: No. So what do you need to do?

AUDIENCE: I need to call somebody.

ANA BELL: Exactly. Who are you going to call?

AUDIENCE: Factorial.

ANA BELL: There you go. What is your name going to be?

AUDIENCE: Factorial at-- is it 2 now?

ANA BELL: Yes, we are at 2, exactly. So you are factorial. Your name is also factorial. And you are going to be called with n is equal to 2. Again, now I have three factorial calls. They're all to the name factorial, but they're all independent function calls. So your job is to return 2 times factorial of 1. Do you know what factorial of 1 is?

AUDIENCE: Yes. Wait, no.

[LAUGHTER]

ANA BELL: As a human you do, but as factorial you do not. What do you need to do?

AUDIENCE: Call her.

ANA BELL: Call her, exactly. Here you go. Your name is also factorial. You can stand beside our lovely other factorials. So your job-- audience, I've already given it away. Your last job is to return 1. OK, excellent. So here is your return value. Now, factorial of 1, are you going to return that value to me? Which one will you return it to?

Exactly. So factorial with n is equal to 2 can now replace their factorial one function with 1. So what is your return value going to be, factorial of 2?

AUDIENCE: 2?

AUDIENCE: (WHISPERING) I got it right.

ANA BELL: 2, exactly. So where do you pass your value along to? OK, now, one thing we forgot, as soon as you made the return value, you disappear.

[LAUGHTER]

You had a very simple job. I'm sorry, but it was really important. You were our base case. Without you, we would have had infinite recursion. OK, so you've passed along your value. So as a function that's done its job, what do you do? Disappear, exactly. Thank you. All right, factorial of, where are we, 3, exactly, what are you-- what is your value going to be now?

AUDIENCE: I'm a 6.

ANA BELL: 6, exactly. So here's your return value. Do you give it to me or-- there you go. Exactly, very good. We disappear. So we've got three function calls that disappeared as soon as they return to value. And finally, 4 times 6.

AUDIENCE: 24.

ANA BELL: And who do you give your value? Me, which I just gave you. Sorry, yeah, that was confusing. Thank you so much, you guys. That illustrated a couple of things. You guys can head back. Thank you so much.

[APPLAUSE]

So we illustrated a couple of things here. I'm going to-- I can do it on the slides as well, just to bring the point home. But let's go through it. So I've got factorial 4. Every time I make a function call, even though it's the same name, all factorial, it's a completely separate environment. Happens to have the same name, but they're just in charge of doing their own job. So here I've got factorial 4 calling 4 times factorial of 3.

As soon as I see factorial of 3, this creates another environment. This is going to be returning 3 times factorial of 2. Again, another environment. This returns 2 times factorial of 1. And a final environment. Our most important environment is that last one with the base case. It allows us to kick start our conquer step. So this base step will return the value 1 to whoever called it. Again, we're not skipping around.

We only return the value to the function that called us. And I know it gets really confusing because everything is called fact in this particular case. But we just have to remember which function called us. And so we return the 1 back up here. This becomes 2 times 1. And they can finish their job. So notice at this point, we've got-- we were 1, 2, 3, 4 functions just kind of hung up and waiting for values to be passed back to us. But now we can finally finish our jobs one by one.

So this one returns a 2. This one returns the 6. This one returns the 24. And the 24 gets printed out. So big idea here, we've got each function call, even though it has the same name, is completely separate, completely independent environment with their own parameters. Those parameters can change within those environments. And that's totally OK. They won't interfere with any parameters in any other environments.

All right. So let's do the Python Tutor link. And then, again, we can just do one more time just to show you what this looks like in terms of the Python Tutor. So here I've got my factorial with n is equal to 4, calls n is equal to 3, calls factorial with n is equal to 2, calls factorial with n is equal to 1.

At this point, just like with the multiplication, I've got all these factorials in the works, but we can start returning values back to whoever called us until we get back to the original one, the original function call. OK, so this is another recap of the observations that we've seen. Each different function call has its own environment. The variables within these environments are specific to those environments. They don't interfere with each other.

And the flow of control-- so when we make a function call, all we know is the function that we call next. We don't skip around. All we know is who we call next and who we need to give the value back up to.

One last thing I wanted to point out. So here I've got the code for factorial, the iterative version and the recursive version. So the one on the left is-- sorry, the one on the right is what we already wrote, so it's factorial recursive. And the one on the left is the iterative version. So I personally think the one on the right is more readable because it's very similar to the way that we would write the expression mathematically.

But if you had a little bit of time to think about it, you can just as easily come up with code that does the exact same job iteratively. So remember, in iteration, we've got our loop. There's no more function-- no other function calls. We have a loop that iterates some number of times. There's some sort of loop variable or loop counter. And there's a state variable that keeps track of the answer of interest-- in this particular case, the product from 1 all the way up to and including n.

So I want to end today's lecture with just a couple of observations. So today we saw some really simple examples of recursion. But I think it outlined some really, really tricky ideas that people usually have trouble grasping when you first see recursion. And that's because you basically write a function in terms of itself. And that can be a little bit confusing. So of course, we applied recursion to some really, really simple things. We did multiplication, and we did factorial.

Depending on how you feel, the recursive version or the iterative version might be more intuitive for you. And certainly for these examples, you did not need to write them recursively. There's a lot of code out there that you actually don't need to implement recursively. The iterative solution is far more intuitive, especially since you guys were first introduced to iteration. You introduced for loops and while loops back in lecture three or something like that. So if that's the first thing you saw, that's usually the first thing that's going to be your go-to.

But there are several problems that are more intuitive to write using recursion. So a couple of examples where recursion is more intuitive is any time when you need to repeat a task for which you don't know how deep you need to go, in which case the recursive calls will take care of making calls to itself to itself to itself to itself until it reaches the base case. You don't need to think about that in your iteration.

So an example of that is this kind of classic one where we have a file inside a file system. If we're looking for a P set, pset.txt, we can have a student who's pset.txt is straight under their user/pset.txt folder. But we might have another person who's pset.txt is going to be within their users, their documents, their schools, their MIT, their classes, their 6.100L, their P sets, their pset1/pset.txt.

So that uncertainty for how far deep you need to search the file system in order to get to the file of interest is a perfect place to apply recursion. Another one is where you have an expression. If you're building your own calculator in code and you have order of expressions-- sorry, order of operations using parentheses, again, you don't know how many parentheses you might need to have a loop go through in order to get to that base expression to figure out the one that you need to do first.

And so that's another case where using recursion is very useful. So in the next lecture what we're going to do is a recap of recursion using another example, a Fibonacci sequence. And then we're going to start looking at recursion applied to lists. And specifically, if we have lists within lists within lists within lists and we don't know how many nested lists we might have, recursion is going to be a perfect example for that.