[SQUEAKING]

[RUSTLING]

[CLICKING]

ANA BELL: All right, everyone, so let's get started. Last lecture, we introduced functions, and we saw some syntax around how to create functions. But mostly, we were interested in motivating functions as a way for us to start writing really clean code, code that's easy to debug, and code that's easy to read in the future. Today, we will continue our fun adventure with functions. And we'll see what it means to treat functions as objects.

So let's recall the example we talked about last lecture. We created this function, is even. So the syntax for creating a function is basically the keyword def tells Python we're defining a function. We decide what name to give our function. Parentheses tells Python in here we're going to name all the arguments, all the inputs to the function. The colon starts the body of the function.

The first part, it's not required but should always be in there as a way for us to implement abstraction. It's called the docstring. So this in green is the docstring. Triple quotes starts our docstring, and triple quotes ends the docstring. And you think of the docstring, also known as the specification, as just a really long comment. And in it, it's-- and the docstring is kind of-- I called it a contract between the person who writes the function and a person who uses the function.

And in the contract, the person who writes the function basically says, this function is going to take these inputs, and I guarantee this function to work correctly when you give me these inputs of these types and these restrictions on them, things like that. And then you also state what the function is going to do. And then you also state what the function will return.

In this particular function, we have only one line. This is the body of the function. But you've hopefully seen functions that are a little bit longer as you did the practice from last lecture. And the body of the function itself-- so the lines of code are basically lines of code that we've seen before. There's nothing special about that except for lines that start with a return. So lines that start with a return basically tell Python that as soon as I see this line with a return, hit.

And when I'm executing my function, I need to stop executing this function, take the value associated with this return and pass it back to whoever called me. A function always returns something. In this particular case, the function will return either True or False, a Boolean. But you can write functions that return integers, floats, strings, things like that. In this case, this is what is returned.

It is possible, and we actually saw this in one of the you-try-its as we were writing our code-- it is possible to write a function that doesn't actually return anything explicitly. So here is an is_even function. And inside the body, the only change I've made is I've eliminated the little return keyword. But otherwise, the work that is done is the same. So here I'm just calculating whether the remainder is 0 or not. So this line of code, when the function is executed is replaced with either True or False.

Notice this function doesn't have a return keyword. But all functions return something. So while function is being executed because of a function call, if the function reaches the end of all of these indented lines here, everything that's indented-- if it reaches the end and no return statement has been hit, then Python automatically returns None. So this is the line without a return statement.

You can think of this code as basically behind the scenes Python putting this little line at the bottom that says return None. Now, this is not something that we would ever write. You just do the operations, maybe you print some stuff out, and then you just omit the return keyword if you want to return None from the function. And None is this NoneType, is a value that is of type NoneType. We talked about it back in maybe lecture one or two, and we haven't really used it that much since.

But basically, you think of it as just having the type NoneType, and there's just one value associated with it, None. And usually, we use this value to represent the absence of a value in our code. So let me just run some code first just to show you exactly some of the kind of things you might observe when you write code that doesn't have a return statement.

So here, I promise this is the last time we're going to see is_even. So here I have two versions of the is_even function. So I have one that I named is_even_with_return, and I have one that is named is_even_without_return. They do very similar things. The difference is that this one has a return statement, where I return whether the remainder is equal to 0. And this one has no return statement, but it just prints whether the remainder is equal to 0.

OK, so let's look at running the code is_even_with_return. And as we're doing so, this first function will be a recap of last lecture, tracing through what happens when we make a function call. So I've uncommented this line. And now I'm running line 13. So when Python sees this file, it basically sees this function definition. And this is not code that runs yet. It's just telling Python that I've created this function inside memory.

When I have this line being run, that's when the function is actually being called and actually being run. So i is replaced with a parameter 3. And at this point, the body of the function is executed. So the first thing that we tell the function to do is print the string "with return." So if I run it, you'll see it prints "with return."

Then it calculates this variable remainder, which is going to be 1, because 3 % 2 equals 1. And then I'm going to return whether 1 equals to 0. So that's going to be False. So as soon as we see this return statement, Python returns out of this function call and replaces the function call entirely with the return value. So this entire line after the function call is executed is replaced with False. So I've just noted that here.

We're not doing anything with this return, right? All we're doing is making the function call, and it just kind of sits on line 13. In order to see the result of the function call, we saw last lecture that we actually wrap the function call around a print statement. And function calls in that sense are kind of just expressions. They do some work. Python evaluates them to some value and then replaces that function call with a value.

So if we wrap is_even_with_return 3, this function call, around with a print statement, Python does the whole thing again. I is 3. It returns False. And this line effectively becomes print parentheses False. And we know what that does. It just prints False to the screen. And there it goes. Notice we still did this print statement because as part of the function body, we tell it to do this print. Everyone OK so far?

So now let's see what happens when we run this function is_even_without_return. So very similar. I've just created an extra parameter here-- or variable here just to show you that you can. So this function, is_even_without_return 3 is being run on line 27. So i is 3. This function will print "without return." And then it calculates remainder to be 1.

And then has_rem will be False. So the variable has_rem will have a value of False. And then as part of the function body, we're going to print the value of has_rem, which is False. So this line here will actually print for me "without return" and then this thing, False. And then the function has no return statement explicitly in there.

So you think about it like Python kind of implicitly adds this return None at the end of the function call. We don't add this. I just wrote it there just to show you that Python would add a line such as this when it reaches the end of the function, but you would never add it. So that means that the entire function call is replaced with None. Yes?

AUDIENCE: So when you-- what happens when you put print in the definition versus when you put return?

ANA BELL: What happens when you put print in the definition versus around the function call?

AUDIENCE: Versus when you put return in the definition.

ANA BELL: Versus when you put return in the definition. So that's the next line. So in the next one, if we were to do what we did before, which is let's print the result of the function call, well, Python will do everything we just did. So it'll print "without return." It'll print False. But then it'll additionally print the return from the function call. So if the return is None, this line effectively becomes print None.

So what we end up seeing or what the user would end up seeing if they actually run this program is they'll see "without return," they see False, and then they see this extraneous None in the console. So you'll see probably this in problem set two. You'll probably encounter an error such as this and maybe problem set three. But don't be scared. Whenever you see a None out in the console, it just means you have to be careful about the function that was called. You probably forgot to return something. And instead, we're just printing the correct value within the function but just never returned it. So that's just something to be wary of.

AUDIENCE: So should you always use return instead of [INAUDIBLE]?

ANA BELL: Yeah, so that's a good question. Should you always use return? It depends on what you want the function to do. Most functions are useful because they go off on their own. They do a task, and they get a value at the end. And they pass the value back to whoever called it. And then you can use that function with many different inputs to give you many different outputs. So usually, you'd want to make functions that return something that you can then do something else with further in another part of the program.

The print within the function should usually be maybe for debugging or for maybe the status of the function, what part it's executing or something like that.

AUDIENCE: OK. And then when you run the function, then it will give you the return. But if you run the print in the function, that's when it does the None?

ANA BELL: Exactly. If the function is not returning anything, then it'll print None. Right, yeah. But if the function is returning something, it will print-- if you wrap it with a print, it'll print whatever got returned. OK. So let's have you work on this. Actually, there's nothing to write. But think about it. So I've got four lines of code here-- add 1, 2; print, wrap that around the print statement; mult 3, 4; and then add that around a print statement.

So try to trace through and tell me what outputs each function call will give me. So add 1, 2, what happens? What do you think the output of this function is? What gets printed to the screen? What is it?

AUDIENCE: Is it addition? [INAUDIBLE] 3.

ANA BELL: Am I telling it to print anything? That's the question. So nothing is actually printed to the screen. Because in the function call add comma 2, we basically map the parameters one at a time. X is 1, y is 2. That was good. We return 3. And so this entire function call is replaced with 3. But we never told that line of code to print that result. So there's nothing printed in this case.

Well, what if we wrap this in a print statement? Is anything printed in this case?

AUDIENCE: Yes.

ANA BELL: Yes, what is printed?

AUDIENCE: 5.

ANA BELL: Yeah, exactly. The add itself gives me 5. And so I'm telling it to print 5. What about the next one, mult-- what is that, 3, 4. Is anything printed as a result of running this line?

AUDIENCE: No.

Yes.

ANA BELL: I heard some yes, some no.

AUDIENCE: The print is in the function.

ANA BELL: Yeah, the print is in the function, exactly. So just because it's a function call doesn't mean we don't print anything, right? We need to check out what the function is actually doing. So in mult, x gets mapped to 3, y gets mapped to 4, and the function body itself says to print the result. So this will print as a part of the body-- prints the 12.

Anything else it prints? No. And lastly, what if we put a print statement around the mult 4, 5? What will that print?

AUDIENCE: 20 then None?

ANA BELL: Yeah, exactly, 20 then None. So the mult itself is going to print same as there. It prints the 20. So the function call returns None. So this entire function call basically is replaced with None, and the line then becomes print None. So this will print the None to the screen. So there's actually four-- four print-outs generated from these four lines.

The first one generates nothing, but the last one generates two lines of print-outs. Any questions about this example? Yes?

AUDIENCE: Can you go over why it prints out None?

ANA BELL: This one here?

AUDIENCE: Yeah.

ANA BELL: Yeah, so the mult, check out what it's doing. It's doing a print statement. So that 20 gets printed out to the console. But what's the return value of mult? There is no return, right? So if there's no return, Python adds the None. That's just something that's implicitly done. So the return from mult because it doesn't actually have an explicit return is None. So we're asking it to print the return, which is None.

OK, so a couple of words on return versus print. So the return only has a meaning inside a function. So as an example, if I just have this file open and I have return 5 just randomly that's not within a function definition, already I'm in trouble. You see that red X. And if I run that code, Python gives me a syntax error. This one's pretty easy to debug. There's a return outside of a function. Yep, there it is.

So return only has a meaning inside a function. It basically says this function has done some work for me, and it's returning back this value. Print statements can be put wherever you'd like, inside functions, outside functions, wherever you'd like. And they all get executed. You can have many return statements inside a function. Like if you have a function that returns 0 if some condition applies or 1 if some other condition applies, then you can have those two return statements.

But as soon as Python during execution hits one return statement, it immediately ends the function, takes that return value, and pops it back to whoever called it. So it's not going to run more than one return statement. Print, on the other hand, you can run as many print statements as you'd like inside the program. And they can all be hit, and they can all generate some sort of output to the console.

So the return statement has a value associated with it. So return 5, return-- we had remainder equals 0, whatever, there's the associated value with that return statement. That value is what gets passed back to whoever called the function. The print statement also you can think of it as having a value associated with it. That's the thing that gets put out to the console. But that value associated with the print statement is just something that's outputted to the console.

It's not being passed around through the program at all. It's just kind of static. It gets put to the console, and then that's it. Nobody else can really use that value unless it's a variable. And then you're just using a regular variable. The last thing I want to show you-- this is kind of cool.

So if we have a print statement just in here, and we run it, obviously that prints that to the console. But what is this print? It's a function, right? It has all the telltale signs of a function. The name is print. The parentheses are there. And I'm giving it one parameter, 5. So if I print the return of the print function. So if I wrap my print function in another print function, what do you think this is going to output?

I'll run it. It outputs None. So the first 5 is due to this. This shows up on the console. But print being a function, it doesn't actually return anything. It does something useful, like take whatever you want and show it on the console, but it doesn't return anything back to whoever called it. And so if I wrap my print function around another print function, I'm basically printing the return of the print function, which is None.

So that's where the second None comes in. All right, so thought of another way, you can make a variable a equals print 5. And if I print a, basically we're saying the return of that first print function is just None. Yeah.

OK, so I'm going to have you work on this code for a little bit. Nothing to write, but there is something to fix. So here's a function called is_triangular. It takes in one parameter, and it's a number, an integer greater than 0. I want this function to return True if n is triangular and False otherwise. So triangular just means it's a whole number such that it's equal to 1 plus 2 plus 3 plus some summation like that.

All right, so 1 is triangular, 3 is triangular, 6 is triangular, and so on and so on. So take a look at this code. It's on line around 49ish. So start by running it, seeing what you get, and I'll give you about a minute or so to see if you can try to fix it. Make sure it runs with all these test cases here.

OK, what's the first thing you should do when you're asked to fix some code that's buggy? Yes.

AUDIENCE: Could you [INAUDIBLE]?

ANA BELL: We can do that. But first, let's run it with something. So let's run it with the first one. Print is triangular 4. So we know the answer should be False. I mean, I told you, so that's good. Yes, it does give me False, which is good. But it also prints out a None. What does that mean for us? Yes?

AUDIENCE: There's no actual return statement [INAUDIBLE].

ANA BELL: Yeah, exactly, perfect. So there's no actual return statement. Like I mentioned with the is_even example, if you're seeing some Nones show up in places, check your returns. So is this function actually returning something? No, it's just printing the result. So it's printing the right thing in this case. So let's start by changing the prints to returns. Yeah.

AUDIENCE: When I run it, it says True and then False?

ANA BELL: For this one?

AUDIENCE: Oh, nevermind. I think I just ran [INAUDIBLE].

ANA BELL: OK. All right, let's run it. Perfect. Yeah, so that seems to have fixed it. What should we do next? Yes?

AUDIENCE: You check the rest of the print statements if it doesn't work for one.

ANA BELL: Yes, exactly. Let's check the rest of the print statements. So the second one, 6, is triangular. So that prints True. And last one, as you mentioned, is going to fail on us. It prints False, but 1 is triangular, right, because 1 is just the sum of 1. So do you know what a fix could be?

AUDIENCE: Change the range to n plus 1?

ANA BELL: Yeah, exactly. So you've spotted it. The range should be n plus 1. If you didn't spot that right away, as I think somebody mentioned there, the first thing we should do is just start putting some print statements. And inside the loop is a great place to put a print statement. We can see what thing we're iterating over. And so if this was still n and we didn't manage to fix it, and we run it, we see that we've iterated when i is 0 right here, and we never actually hit 1.

So the fix for that is make sure we go up to n including n. And now if we run it and remove this print statement because it might be a little confusing, that now gives me the correct answer. Last step should probably be to run the other two cases again, just in case my fix broke something else. And it didn't. The other two cases are still the same. Questions about this code? Does it make sense?

OK, so now, last lecture, I mentioned that once we write functions, it's really easy to include these functions in larger pieces of code. And it makes those larger pieces of code very nicely readable. So let's try to do the same with a slightly more complex example. Let's try to create-- take our bisection_root code and make it into a function. And then there's going to be an exercise in a couple of slides where you get to use this function.

So inside of this function here is basically what we had like three lectures ago. It's just the bisection square root code. The only thing I've done is I've wrapped it around a function definition. So def, I gave it a name-- bisection_root is a pretty nice name-- and figured out what input this function should take. So the input should be the x I would like to approximate the square root of.

One thing I didn't do is put a docstring on this, so that's my bad. But the docstring would say x is a positive integer greater than 1 and returns the approximation to the square root of x or something like that. OK, so here we're hard coding epsilon to be 0.01. We've got our low and high endpoints, just remembering what the bisection_root does. And we're starting out with a guess that's right in between the low and high.

The while loop here is going to do the work for us. So the while loop condition is while the difference, the absolute value, the difference between our guess squared and the actual x we're trying to find the square root of is bigger than epsilon. So while we're still farther away than epsilon, we have more guesses to make. The way we make the guesses is by updating the low endpoint or the high endpoint, depending on whether our guess was too low or too high. This should be review hopefully.

And then after we've decided on which endpoint to update, we update our new guess to be whatever high plus low is divided by 2 again, so the midpoint of those where either high or low would have just changed right because of this if else. And this loop will just keep going over and over and over again, making better and better approximations until we come within plus or minus epsilon of the square root of x.

The difference between this code and what we wrote a few lectures ago is this part down here. So a few lectures ago, all we could do really was write a print statement where we took our guess that we ended up with and we printed it along with that guess is close to the root of our original x.

But instead, since we're writing a function, I would like to take the result, my approximation to x, and return it. So somebody can call this function many, many, many times with different values of x and figure out a bunch of different approximations for all of these different x's. So here I have the function calls. So I've got bisection_root with 4 and bisection_root with 123. And then I can just print the results of these.

So here is the bisection_root function. I've got my printout commented out because I don't actually need it. The rest of the code we'll do something useful with the approximations. So in this case, bisection_root of 4 gave me 2.0. So that's the approximation. And the bisection root of 123 was approximated to 11.09.

OK, so what I would like you to do, and this is going to be a little bit involved code, it will require some thinking, is to write a function called "count the numbers with the square root close to" n plus or minus epsilon. And I'll help you out by drawing something on the board. But I would like you to do the code for it.

So the idea here is that you have some n that's given as an input. And you have an epsilon that's also given as an input. What you'd like to find out is how many whole numbers have their square root within plus or minus epsilon of n. So this is kind of hard to wrap your mind around without actually drawing a picture. So this is also something you should try to do in quiz situations, p sets, things like that.

Don't code right away. Try to draw a picture kind of depicting what we're asking for here. So here, we'll start with a line. This is our number line because we're doing the square root. We want to know how many integers have a square root with an epsilon of n. So let's start with an n. And we have something plus or minus epsilon. So this is epsilon, and this is also epsilon.

In the end, we want to know how many integers have a square root of i-- so actually, I'll do it like this. Square root of i is equal to somewhere in this range. Does that make sense so far? That's what we're trying to figure out. The square root of i is somewhere in this range. So that means i is going to be some giant number out here. So this line can go further out.

So in the example here, I've got n is equal to 10. So I know for sure that an i of 100, just kind of us as humans, would work because the square root of 100 is probably going to be approximated to pretty darn close to 10. So I know that that value will be within plus or minus epsilon of 10. But there's probably a couple numbers around 100 that also match this criteria.

If I take the square root of 99, according to this example, that approximation puts me within plus or minus epsilon of 10. So it's going to be-- square root of 99 is going to be like 9 point whatever is here, .95. So that's within plus or minus epsilon.

And similarly, square root of 101 and 102 also work because if I take the square root of these guys, that will also put me within plus or minus epsilon of 10. So the goal here is basically to figure out these numbers, 99, 100, 101, and 102.

You should use the power of computation and computers being able to do a task really, really quickly to basically say I'm just going to brute force my way through this problem and say I'm going to test each number, one at a time, all the way up to some pretty large number. So you want to make sure you hit 99, 100, 101, 102, maybe going up to maybe n cubed.

If you go and take the square root of some i cubed, you know you're going to hit all these values within plus or minus epsilon. So you're just going to brute force, look at all the integers between 0 and n cubed, and figure out if this square root-- the approximation to this square root is within plus or minus epsilon of n. If it is, keep a counter and increment it. And if it's not, ignore it. And that's the idea to this question.

Loop and a check, that's it. And you can definitely feel free to make use of the bisection_root function that we wrote in our code. You should definitely use it because it's very helpful. So around line 96 is where you can write your code. All right, does anyone have a start for writing this code? Or how would you think about it? Yes.

AUDIENCE: For i in [INAUDIBLE] n cubed?

ANA BELL: For i in range n cubed, yeah, we can do that. All right, so this will give me numbers 0 through n cubed. Perfect. So I've generated basically this sequence now. What do I want to do once I have i? And you can always write a little comment for yourself what you want to do once you have i. So in English, what would you want to do once you have a number like this?

Take the square root, yeah. Take the square root of i. How do you want to take the square root of i?

AUDIENCE: [INAUDIBLE].

ANA BELL: We can. Shall we use our bisection_root? We can too. Yeah, we can do both. So let's use the function we just wrote. So bisection_root of i, this gives me square root. So now sqrt is going to be some value here. It could be 10. It could be 99.5. It could be 99.7. What do I do with this number now?

Yes?

AUDIENCE: Could you use an if statement [INAUDIBLE]?

ANA BELL: Yes, exactly. Let's use an if statement. So if-- and there's many ways we can use the if statement. We could do absolute value. That's what we've been doing already. So if we take n minus the square root-- so n minus this value we just calculated is less than epsilon.

So here we know that square root is within epsilon. And what do we want to do once we know that the square root is within epsilon? Well, if we don't know, we can look at the docstring. So we need to return how many integers have that square root with an epsilon of n.

AUDIENCE: [INAUDIBLE].

ANA BELL: Yeah, exactly, keep count of it. So count plus equals 1.

AUDIENCE: [INAUDIBLE].

ANA BELL: Yes, and I do have to initialize count. Count equals 0, right before my-- OK, anything else?

AUDIENCE: The return function?

ANA BELL: Yeah, we do need to return. So at the end of the loop, we can return our count. OK, run it. What is this from? Oh, this is from the other two lines here. So four. I think that works because from the example there were four numbers that worked.

To double check, we can-- or if something went wrong and the number you got wasn't what you were expecting-- again, print statements very useful. So we could print the value of i, so this thing here we're trying to find the square root of. And we can print the square root of that value. And so if we actually add it to the print-- to the code here, we see the four values that we grabbed, 99, 100, 101, 102.

And now that we wrote this code, we can actually make really simple changes to it. And we have some pretty useful code. So if we make our boundary bigger, 10 plus or minus 1, we're going to get more values that match this criteria. So in fact, we got 40 of them, all the way from 81 all the way up to 120. They all match the criteria, which is when you take the square root of that value, it's plus or minus 9 to 11.

Any questions about this example? I know it's kind of involved. But I hope that actually drawing a picture helped explain what we were trying to get at. And then at that point, it should have been pretty easy to figure out the structure of the code itself. Any questions? Yes?

AUDIENCE: A question regarding the range. Why does it have to be that large number? It can be smaller.

ANA BELL: It could be smaller, yeah. I mean, we could have done n to the power of 4. We just couldn't do n squared because then we might miss-- well, we would definitely miss 101 and 102 in that particular example. And in fact, if our epsilon is really big, we might-- actually, I'm not sure about the math, but if our epsilon is really big, we might actually need to go bigger than n cubed as well. I'd have to think about that. But we just-- it's OK. I mean, it's fine to make it big. It doesn't take that much longer to compute because running the function is very quick to Python anyway. Yeah, there's a question.

AUDIENCE: I had a similar question. So is there a reason why we chose n cubed as the arbitrary number that's big enough?

ANA BELL: Yeah, arbitrary number that's big enough. What we could have also done just along those lines is we could have done something a little bit smarter in here, where once we find a number that actually works, like once we start incrementing our count, we could have some sort of flag that keeps track of as long as we're incrementing the count, keep going. But at some point, you know you're going to reach a number that's too big.

And at that point, you can just end the function early. You can just break out of the loop, and you don't need to keep looking all the way up to n cubed. So we could have done something a little bit smarter to make the function just a little faster with flags, which you can try. So see if you can have the program stop as soon as you hit 103. See if you can write the program that uses a flag to trigger that event. And then when that event is true, just break out of the loop or return immediately or something like that.

Other questions? OK, so let's zoom out a little bit on functions. We did this a little bit last lecture. This is a function that we actually wrote last lecture. It was sum of odd numbers between a and b. This was essentially our black box. Remember that now that we're writing functions, we are kind of separating ourselves as a programmer who writes a function-- you basically make this nice modular piece of code that can be reused over and over again.

So we're separating that aspect from somebody who's using a function. So once there's a function already written for you, you just use it in code, like we used the bisection_root here. I know we wrote it, but I guess technically I wrote it. But here we just kind of used it. And we used it to write this nicer, more complex piece of code. And so this is what we do. We basically create this black box. And once the specification or the docstring of the black box, you don't need to know how it's implemented in order to use it.

But what I wanted to mention is something I mentioned last lecture is the function definition is just creating a function object inside the memory. And the name of this function object is the name of the function. So if we're thinking about the program, there's the orange box, we have an object that just happens to be a function, which has some code associated with it, whose name is sum_odd.

And kind of drawing a parallel to that is when we create just a variable, as we have been so far. Here we're creating an object 2 whose name is low. So in that same way, that black box is basically saying, I am creating a function object that has some code associated with it whose name is sum_odd. So in this case, I've got sum_odd, low, and high as three sort of objects inside my program.

And then only when I make a function call does the code associated with the function object run. So when I'm defining the function, it does not run. It just stays inside computer memory as an object that exists. And when I make my function call is when I use that object. So the function call basically takes my variables and matches them to the function definition.

So a gets matched to low, and b gets matched to high. And low and high in the function call have actual values associated with them, 2 and 7. And so that function will then go ahead and do the work. And at the end, it's going to return something, either an actual value or None. And then that actual value replaces the entire function call. So in my program, the variable my_sum here is going to be equal to the return. Just a little recap, but hopefully this kind of keeps bringing that point home.

So now we're going to talk about in more detail what exactly happens when we make a function call. So when we make a function call, you can think of the program as sort of taking a pause. I've got my main program, and in my main program, I have a function call. That main program will just pause for a bit. And that function call, you can treat it as sort of a little mini program that needs to run and terminate, return a value, before the main program can resume executing.

So that little mini program, that function call basically creates its own little environment that it lives in. So in that little environment, it can create variables just like we would in a regular program. It can modify variables. It can print things. It can do all this work within its body. And at some point, it'll finish its job, finish its task, and it'll have some value that's the result of all of that work that it did.

And that value is what it hopefully returns back to the main program. And then the main program can finish its-- can finish its job. So what's key here is that every time you make a function call, you basically create a new environment. And that environment is completely separate from the main program environment. As soon as the function call terminates, that function call environment disappears. So any variables that were created within that environment of the function call will also disappear. So all we're left with is just what's in the main program.

So now we're going to talk a little bit about environments. And if you understand this, you'll understand 80% of functions and what to do with them. So basically, when you first run your program, the program enters what we call the global environment, the main program environment. And any time you make a function call, we're creating this new environment. So what exactly happens when we create these-- when we do these function calls? How do these environments interact?

And the answer is they don't actually interfere with each other that much. They only interfere with each other through passing in parameters and through returning values. But beyond that, these two different environments, the main program environment and a function call environment, can actually have variables that have the same name but don't interfere with each other because they exist in different environments.

So we're going to look at this example to showcase that. So here's a function. It's pretty simple. It does not do much. It takes in one parameter, probably a number, and adds 1 to it. So it takes in an x and does x plus 1, reassigns x to it. And then it does this print statement and then returns the new value of x. So it added 1 to whatever you passed into it, and it returns that new value.

So that's the definition. Again, this just sits in Python memory. It doesn't actually get run until we make a function call. The parameters here when we wrote our function are called formal parameters because there's no actual value associated with them. We're writing this function assuming that at some point we're going to get a value for x. But at the time we're writing the function, there's no value for x. It's just this abstract variable.

And we're using that variable x within the function body, assuming that at some point we're going to get an initial value for x. So x is equal to 3, at which point the body can then execute. Now, when you make a function call in the main program scope, that's when you pass-- make a function call with an actual parameter. So here, you'll notice I'm using the same name x. But this x inside the main program is different than the x that's this formal parameter of the function.

This actual parameter, when we make the function call, is mapped to the formal parameter. So at that point, the formal parameter can get the value of the function call, which is 3. And in fact, it doesn't actually matter what we name this variable out here. We can name x is equal to 3 and make the function call f of x. But we can also have y is equal to 3, and we make the exact same function call, f of y, because we want to pass in 3 as a parameter to this function call.

So this x out here is different than this x over here. So the-- oh, yeah, go ahead.

AUDIENCE: Which one is the formal?

ANA BELL: The formal is the one from the function definition. We say it's formal because there's no value associated with it when you first write the function. You write the function first. There's nothing going on here. And then you have some code that actually now is taking on some values and you can run it.

So let's trace through this code little by little to see exactly what environments get created as we make function calls. So again, this is my black box. It's a function. When I first run the program, we finished the function definition. So we're at this point in our program, right before we do x is equal to 3. Inside my computer Python memory what I have is one environment created. And that's the environment of the main program. The only thing I have in this environment is my f because at this point in the program, where the red arrow is, I just had a function definition.

So again, it's a definition-- it's a function whose name is f, and it's an object. It's not being run quite yet. It's an object that contains some code. Now we have x is equal to 3. So that's pretty easy. Inside my main program environment, I've got a variable name x, whose value is 3. And then I have my function call.

So as soon as Python sees a function call, it creates a new environment. And the current environment, where the call is being made from, so the main program one, will be put on hold. So here I'm calling function f. So now I'm creating this new environment that-- think of it like this mini program, this little task that needs to get done before the main program can continue executing. So I need to figure out what's going on in this mini program, in this function call to f.

All right, so here's my new environment, the scope of f. The first thing that we need to do is figure out, what are the parameters of f? So we look at the function definition, and we see it has one parameter named x. So we're going to take that x, and the first thing we're going to do is map the formal parameter to the actual parameter. So we're going to make the formal parameter of f named x take on the value 3.

That's kind of what we've been doing already. But now this is getting down to details, just details. We've mapped all the parameters. The body of the function executes. I've, again, kind of blurred out this one because we're not in this global scope. We're trying to figure out what f is doing. So the body of f says take x, add 1 to it, and reassign it to x. So what's x inside my function? It's 3. We add 1 to it, and we make x be 4.

I skipped one thing, which is if in my main program I had y is equal to 3 and f of y, nothing really would have changed. My formal parameter of f is still x. And I'm still mapping x to the value that's in my-- here, in the actual parameter.

So in my scope of f, I've got x is 3. I increment it by 1. It gives me 4, and I resave it back into x. And again, there's no collusion-- there's no collision here in terms of naming because the scope of f, the environment f, has a variable named x, and I'm just doing stuff with the x that f knows. I do have another x inside my global scope. But that one's put on hold for now. All right, so I've done x equals x plus 1. Then I do the print statement.

So in f of x, x equals 4, that gets printed out. And then I return x. So the thing that gets returned is the value of x, so 4. And this again replaces the function call. So this gets returned back to whoever called me. And the environment that called me was just my main program. And here I'm going to return 4. And this is going to replace that with 4. As soon as the function sees the return and returns that value back, it goes away. So notice that x that we had created is gone.

Now we're in the main program, there's no confusion. My main program has its own x. That other x that was part of the execution of f is gone because that function finished its job, and it doesn't need its environment anymore. So now the return of the function replaces f of x, and we see z is equal to 4. OK, that was super detailed. But that's kind of what happens step by step when we make a function call with the new environments being created.

So if you can understand that, it should be-- it should be pretty straightforward, and you won't get confused when you see an x out here. You have f of x as one function and then maybe another function that has g of x and so on. So in order to know the scope that you're in, the environment that you're in, you need to know what expression you're evaluating. So here, we were evaluating this function call. So that means that we were inside the environment of f.

Another example, and this one's a little bit weird. It shows some of the nuances of Python. And these aren't necessarily true in other languages. So I'm just going to do the drawing of the scopes out here. So let's start with the one on the left. So you can see here I've got one function f of y. And I've got the main program that creates x is equal to 5 and then a call to y.

So inside my main program, I've got x is equal to 5. And then I have a function call to f. Function call means we need to create a new scope. So this one's put on hold for now until we figure out what f parentheses x is right here. So the first thing we need to do is grab f and take all the formal parameters of f-- there's one; its name is y-- and map them to the actual parameters. So I'm calling f with 5. So I'm going to map y to 5.

This function is going to take-- now the body of its function, x is equal to 1. So it creates also an x whose value is 1 just within its scope. It adds 1 to x, so this becomes 2. And then it prints x, so it's going to print 2. And then the function terminates. It returns None. There's no return statement. And the function is done.

So this line has now finished. And the last thing that the function does after it's done the return is the scope goes away. And the last thing we need to do now is print x. So this will print the value of x in the global scope, which is 5. So the output of this little piece of code on the left side here is 2 and 5.

OK, what about the middle code? Similarly, I've got a function definition, and then I create x is equal to 5. And then I make a call to g. All right, x is 5. So as soon as I see a function call, I need to create a new scope. And I need a map all the formal parameters of g. It has one formal parameter. Its name is y. That one will be mapped to whatever I made the function call with, 5. X is 5 out here.

So that gets mapped to 5. What is this function going to do? Well, it prints x. What's x inside the scope of g? Do I have a g inside x-- an x inside g? No. So this is something that Python does. It says, well, if your environment doesn't have a variable named x, in this case, look further out and see who called you.

Well, which environment called this g? The main one, right? Does your bigger environment, the one who called you, have a variable named x? It does, right? It's 5. So Python grabs the value associated with that larger environment. And if that larger environment didn't have one, it would look further out and further out, out until it doesn't have an environment to look at.

So g is going to print the value of x, which is 5. And then it's going to print x plus 1, which is 6. And then it's done. It returns None. And then as soon as it returns None, this scope goes away. And what we're left with is the global program, and we print x, which is still five. What I want you to notice is that that function g printed x plus 1 but never modified x.

We never said something like x is equal to x plus 1 or something like that. We just figured out what x plus 1 was and printed it. All right, one more example, and this one will actually end up in an error. So here I've got x is equal to 5, just like before. And then I have a function call to h. So again, a function call means a new scope is created.

I've got one variable, y. That's my formal parameter. It gets mapped to whatever I call the function with, 5. Oops, that's an s. And then what is this function doing? That line, x plus equals 1 is x is equal to x plus 1.

This is actually an error. Python doesn't let you do that. And the error it gives you is actually what it says there, so UnboundLocalError, local variable x is referenced before an assignment. So it doesn't actually grab the value from the outer scope, like we did in the middle bit. It doesn't grab it because it thinks you're trying to create a variable named x inside h. And you're trying to add 1 to x. But you never had a line that said x is equal to something originally inside h.

And so when you're trying to say x is equal to x plus 1, it's trying to look for an x inside the scope of h, but it doesn't have one. And so that's where we get that error from. And this is not-- I mean, it's just a nuance of Python, but it's kind of important to understand that you can access variables, but you can't change variables outside of your scope. So the middle one just accesses a variable, adds 1 to it, and prints it. But we never said x is equal to this value.

And it's kind of like, I guess the error you get is kind of like if you made this be something completely different, like z. You would get the same error. It would be error variable z referenced before assignment. So you can get x plus 1, but I don't know what z is. This should be z.

AUDIENCE: So could you say, like instead of the definition, like z equals x, and then do x plus equal 1? Because you're taking it from outside and--

ANA BELL: So if you-- no, because if you want to-- if you want to explicitly say that you're taking it from outside, there's a keyword called global that you would need to write that explicitly says, hey, I'm grabbing this variable that is not part of me. It's part of-- it's in the main program, the global scope.

OK, the last thing I want to talk about is using functions as arguments to other functions. So the way I've sort of been explaining a function definition is basically saying that when we define a function, Python essentially puts some code in memory whose name is the function name. So basically, the function name creates for me an object inside memory that happens to be a function object.

And just to show you what that means is we have a function is_even. We've definitely created it. If we say the type of is_even, it's function. So the function is_even actually has a type, and its type is a function in Python. So functions are basically just objects, just like an integer is an object, a Boolean is an object, a float is an object. A function it an object. It just looks different. It has a bunch of code associated with it.

So if a function is an object, what that means is we can use an assignment operator on a function name. So we can have two names of functions that point to the same function code. We can use a function as an argument to another function, like a parameter to a function. Or we can return a function from another function.

So here's an example. Pretend that this is our code file. We've got the memory. The first line of code here, the definition, basically creates this function object for me in memory. It's kind of like a variable. Is_even is the name of this function object. And this variable is bound to my function object with some code associated with it.

So you think of the function as just an object. Similarly, when we write r is equal to 2, I think of that as the same thing. R is the name, and I've got an integer object whose value is 2. That's exactly what happens when we create a function definition. Similarly, pi is equal to 22 over 7. Pi is the name associated with a float object that has that value.

So what we can do, now that we've established that a function is basically an object with a name, we can say a line like this, my_func equals to is_even. The right-hand side here is just the name of my function. It's not a function call. Notice, there's no parentheses after is_even. There's no parameter, none of that. It's literally the name of my function.

So inside memory what I've ended up doing is I have two-- oops, I have two names, my_func and is_even, that both point to the exact same function object. So that means that that function object, so this is_even function, can be referenced by both of these names. So on the next two lines here, a equals this and b equals this, I'm running the same code just referenced by different names.

So then a is going to be bound to False, and b is bound to True because I'm accessing the same code, fundamentally, by different names. Does that make sense? Yes, awesome. So everything in Python is an object, including functions. It's strange to think, but there you have it. So let's look at this code. I've got three function definitions and only one function call.

What are the function definitions? One, I have named calc. It takes in three parameters. One I have add. It takes in two parameters. And one I have div. It takes in two parameters. Add does something pretty simple. Div has maybe a print statement but also does something pretty simple. Calc is the one that's really strange, right? Because it takes in these three parameters, but what's the thing it's doing in here? It's kind of treating one of the parameters, op, operation, as a function.

That's what's strange about calc. Let's trace through the code to see exactly what that means for us. So when I first run my program, I have three function definitions. So I'm creating three function objects inside memory--calc, a function object that has some code; add, a function object that has some code; and div, a function object that has some code.

And then we get to the good stuff, res equals the function call. So res is going to be a variable. That's going to have a value. What value? We need to figure that out. Calc is a function call. Every time we have a function call, we need to create a new environment. So now we are creating our calc environment.

So we've put aside the main program scope for now. And we're focusing on what calc is going to do. First thing we need to do is take every single one of our parameters and map it to the actual parameters. So the first parameter is op. It gets mapped to add. The next parameter is x. It gets mapped to 2. The last parameter is y. It gets mapped to 3.

Is everyone OK so far? Yes, OK. I've literally just matched names of formal parameter to actual parameter. OK, so now we finished mapping the parameters. Next we get to run the body of the function. Return, what is this? Let's replace op x and y with the actual values. This basically becomes return a function call, add 2, 3. I've just replaced the names. That's it.

What's add 2, 3? It's another function call, right? So calc is going to have to be put on hold because I have to figure out what add is going to return. OK, so what's add going to return for me? Well, add 2, 3 is what I'm trying to figure out. So I'm going to map a to 2, b to 3. It's going to do 5 as the return. So it returns 5 to whoever called it. And whoever called it was calc, right here.

So this expression here, op x, y, which was add 2, 3, is replaced with 5. Everyone OK so far? Awesome. And then calc can now finish. Notice add finished its job. So it went away. Now calc can finally return its value. So it can finish as well. So this one will return 5 to whoever called it, which was the main program. And finally, calc has finished its job, and it returned 5.

So step by step, we just kind of trace through the code, functions out to in, and replacing variables wherever needed. So it's your turn. Tell me what's the value of res given this function call to calc. And what's going to get printed? So we can even write our functions. So in the main program, what do I have?

AUDIENCE: Calc and div.

ANA BELL: Yep, calc and div are my functions. That's it?

AUDIENCE: Is res also there?

ANA BELL: Which one?

AUDIENCE: Res.

ANA BELL: Yeah, res will be the result, yep. And the res we will have a question mark because we don't know what it is yet. And what's the first thing I need to do? Yeah, make a new scope, exactly. So that's the scope of calc. And we're going to map op to div. What do I have? X and y to 2 and 0. Thank you. So what's op going to do?

AUDIENCE: [INAUDIBLE].

ANA BELL: Yes, exactly. We make another scope for div. A is 2, and b is 0. So we're kind of two scopes deep. What's div going to do?

AUDIENCE: Print out a denominator [INAUDIBLE].

ANA BELL: Yep, so div prints out the thing, "Denom was 0." And what's div returning?

AUDIENCE: None.

ANA BELL: None, perfect. So div returns None here to calc. And then div is gone. And then None gets returned from calc here. And then calc is gone. And all I'm left with is res equals None. Exactly, the return of calc.

One more example showing scope, just kind of showcasing these-- sort of the same idea. So I've got three functions here, func_a, func_b, and func_c. Func_a takes in, you can see, no parameters. Func_b takes in one parameter. Func_c takes in two parameters. And if we scan the code, we see that one of them is weirdly doing something. So it's actually going to be a function because you see we're calling it like a function inside here. So we know f is going to have to be a function.

So if we run this program, first three function definitions, basically put some code for us in the memory, when we make func call-- sorry, func_a call, this creates a new scope. A has no parameter-- or func_a has no parameters. So there's nothing to bind. All this function is going to do is print "inside func_a" and then return None. So that whole thing is going to print None.

Next, func_b is going to be another function call right here. So it creates a function scope right here. We map the formal parameter y to 2. And then we finish mapping all the parameters. And what we need to do next is do the body.

So we print "inside func_b," and it just returns the value you passed into it, so not a very smart or interesting function. So it prints that and returns 2 back to whoever called it. Whoever called it was here. So this print statement becomes print 5 plus 2, the return. So that's going to print 7 to the console.

And lastly, the interesting one is going to be func_c. So func_c, notice I'm calling it with an actual function I have in hand, func_b, one of these that I've defined here. So func_c is a function call. So there's my scope. I am mapping formal parameter f to func_b and z to 3. So just mapping one by one. And then I'm doing the body of func_c. So the body says now print this and return this. So we print the statement.

And then the return basically says, well, what's f function call f z? We have to figure out what the actual values are. And it's func_b parentheses 3. So that's another function call, which means another function scope. Again, not a very smart or interesting function, this func_b. It just takes in the 3, it prints inside func_b, and it returns the 3 back to whoever called it.

So that function is done. And then the func_c can terminate and return 3 to whoever called it, which was out here. And notice as soon as a function call terminates and does a return, it immediately-- all of its variables, everything that got created inside that environment go away. They get wiped out.

OK, I'll give you about a minute to try this. So write a function that meets the following specifications. So I have a function named apply. Criteria is a formal parameter. So at some point, you're going to have a function that does this. It takes a number, an integer, and returns a Boolean. So however a function does that, that's what's going to be passed in. And then n is an integer.

And what I want you to do is tell me how many numbers from 0 to n match that criteria. So when I apply the function criteria to numbers 0 through n, how many of those actually return True on that function? So just to show you something-- what this means concretely, here's my function apply. Here's a function that I could call the apply with, is_even. Sorry, I lied. I guess we are seeing is_even a few more times in this lecture.

So here's a function is_even. And basically, I run apply by saying I want to run function apply with the name is_even. So here I'm mapping name to numbers 0 through 10. So I'll give you about a minute to try it out. And then I can write it just so we have some-- so we finish on time. Does anyone have a start?

So we know we want to touch each number 0 through n to see whether this criteria applies to them. So what's the start to get that going? Yeah.

AUDIENCE: For i in range--

ANA BELL: For i in range n plus 1 because we want to include n? How do we apply the function criteria to each one of these values?

AUDIENCE: Criteria parentheses?

ANA BELL: Yeah, exactly. We just say criteria. And this name will be replaced with whatever function we're going to call apply with, i. And this criteria i will basically be the return of criteria. What did I say criteria returns? It takes in a number and returns a Boolean. So we know that this is a Boolean. What do I want to do with this Boolean?

If it's true, I want to count it. If it's not, I don't. So if criteria i count plus equals 1. I'll put this up here. And let's remember to initialize our count. And then that's it, right? If it doesn't match, then I don't care about doing anything with it. So then we just return count.

So notice I'm using my function here. That's just the parameter, kind of like a placeholder for any other function. So this is_even function, when it's a parameter to apply, will tell me 6, right? 0, 2, 4, 6, 8, 10, that's six values that match this criteria.

And what's cool is that I can actually create any function. So if I want a function that's called is_five, for example, it takes in a number and returns True if that number is equal to 5. It's still a function that takes in an integer and returns a Boolean.

All I need to do then is run this apply with the function is_five. So I just changed that here, and then if I run it, it should just give me one value, the 5, of course, is one that matches this is_five criteria between 0 and 10.

Yeah, so that's basically it. So we saw some functions-- a lot more you can do with functions. They're basically objects in Python. So they can be manipulated just like you would any other object. You can have them be parameters to a function. You can have them be returned from a function. You can assign another name to this function body, things like that.

I showed you how to think about environments so that the naming doesn't get confusing. As soon as a function call is made, that means another environment is created. So variables created within that environment have no influence on other variables created in other environments. And functions are a very nice way for us to write code that can be easily be built up on. OK, that's it. Thank you.

AUDIENCE: Thank you.