PROFESSOR: All right everyone let's get started. All right good afternoon on this rainy, rainy sad afternoon. So-- I'm glad we're inside though-- all right so Lecture 4 of 6.0001 in 600. Quick, quick recap of what we did last time. So last time we did a little bit more string manipulations, and then we saw how you can use for loops over strings directly. So instead of having for loops that iterate over range-- so 0, 1, 2, 3, 4, and so on-- you saw that it was more powerful to sometimes use for loops that iterate over string objects directly.

So that was the first half of the lecture. In the second half, we started looking at different ways that you can implement the different implementations to the same problem. So we saw the problem of finding the cube root, and we saw some implementations. We saw the Guess and Check method, and the approximation method.

And then we looked at what I thought was the most powerful method, which was the bisection method. And this one, if you remember, I played a game with someone in the audience where I guessed a number between 0 and 100. And we saw that I was able to guess that number really, really quickly using the bisection method. And that's the method that you're going to implement-- that you are currently implementing-- in your problem set.

OK so today-- so that sort of finishes introduction to some of the more basic mechanisms in Python. And today we're going to talk about how to structure your programs such that you write nice, coherent code-- reusable code-- by hiding away some of the details in your code. And to do that we're going to look at these things called functions.

All right so just stepping back and sort of getting a high-level view of how we write the code so far. So so far the way that you've been writing code for your programs is you open a file, you type some code to solve a particular problem given, like in your problem sets, each file contains some piece of code, you have sequences of instructions that contain maybe assignments, loops, conditionals, and so on and so on. But really you have one file that contains each code and you write everything in that particular file.

But this is OK for smaller problems that we've been seeing so far, but when you're starting to write large pieces of code it's going to get really messy, really quickly. So think about if you want to use a for loop in one part of your code, and you find it useful to use that same for loop in another part of your code.

Some point in the future as you're debugging your code, you might want to change your original for loop, you have to figure out all the other places where you've used that type of for loop for example. So as you're scaling your code, you'll find it harder to keep track of these details. So this is where functions will come into play in today's lecture-- will help you out.

So if you want to be considered a good programmer, a good programming style would be to not necessarily add lots and lots of lines of code, but really to add more functionality to your programs. So how many different things-- how many different features-- can your program do, rather than how long can your code be. And that'll help you later on look at your code if you need it for a future class, and it'll help others if they want to look at your code later on if they find it useful.

So today we're introducing this idea of functions. And functions are mechanisms to achieve decomposition and abstraction. So these are two key words here that are going to pop up in today's lecture and also in future lectures. So before I introduce decomposition and abstraction in the context of functions, let's first take a look at just sort of a real-life example.

So let's take a projector. I'm using one right now. Quick show of hands. If I give you all of the electronic components that are part of a projector-- resistors, a fan, a light bulb, a lens, the casing, all of the different parts in it. Who here would be able to build a projector? Do I see a hand? No? Ooh oh yeah nice! You can also lie. I won't know the difference. But if you can do that, I'd be very impressed.

All right so you can't really put together a projector right? Another show of hands. If I gave you a projector that's fully assembled and I gave you a computer, for example, who would be able to maybe figure out within let's say an hour how to make them work together? Good, a fair bit of the class. That's perfect. That's exactly the answers I was trying to get at here.

So none of us really know how a projector works-- the internals-- but a lot more of us know how to work a projector, just given maybe a set of basic instructions or just intuitively speaking. So you see the projector as sort of a black box. You don't need to know how it works in order to use it.

You know maybe what inputs it might take, what's it supposed to do at a high level. Take whatever's on my screen and put it up on the large screen there, just magnify it, but you don't know how it does it-- how the components work together. So that's the idea of abstraction. You don't need to know how the projector works in order to use it.

OK that's abstraction. The other half of that was decomposition. So let's say that now, given a projector, I want to project a very, very large image down on a very large stage. For example, this is from one of the Olympics. It's a stage of what, like 10 football fields, something like that? Something massive. You could build one projector that's able to project a very large image, but that would be really expensive and you'd have to build this one projector that's used for this one time.

So instead what you could do is you can take a bunch of smaller projectors and feed different inputs to each one of them. And as you're feeding different inputs, each one's going to show a different output. And then you're going to be able to have all of these different projectors working together to solve this larger problem of projecting this really cool image on a very large stage.

So that's the idea of decomposition. You take the same projector, feed it different inputs, it does the exact same thing behind the scenes, but it will produce a different output for each one of these different inputs. So these different devices are going to work together to achieve the same common goal, and that's the idea of decomposition.

So these is where I apply to the problem of projecting large image, or a projector in general, but we can apply these exact same concepts to programming. So decomposition is really just the problem of creating structure in your code. In the projector example, we have separate devices working together. In programming, to achieve decomposition you're dividing your code into smaller modules.

These are going to be self-contained, and you can think of them as sort of little mini-programs. You feed in some input to them, they do a little task, and then they give you something back. They go off and do their thing and then they give back a result.

These modules can be used to break up your code, and the important thing is that they're reusable. So you write a module once-- a little piece of code that does something once-- you debug it once, and then you can reuse it many, many times in your code with different inputs. Benefit of this is it keeps your code organized and it keeps your code coherent.

So functions are going to be used to achieve decomposition and to create structure in our code. We're going to see functions today in this lecture, and in a few weeks, you're going to actually see-- when we talk about object oriented programming-- how you can achieve decomposition with classes. And with classes you can create your own object types like adding some floats. You can create your own object types for whatever you want, but that's later.

OK so decomposition is creating structure in your code. And abstraction is the idea of suppressing details. So in the projector example, remember, abstraction was you didn't need to know exactly how the projector worked in order to use it. And it's going to be the same idea in programming.

So once you write a piece of code that does a little task, you don't need to rewrite that piece of code many times. You've written it once, and you write this thing called a function specification for it, or a docstring. And this is a piece of text that tells anyone else who would want to use it in the future-- other people, maybe yourself-- it tells them how to use this function.

What inputs does it take? What's the type of the inputs? What is the function supposed to do? And what is the output that you're going to get out of it? So they don't need to know exactly how you implemented the function. They just need to know inputs, what it does, what's the output. Those three things.

OK so these functions are then reusable chunks of code. And we'll see in a few examples in today's lecture how to write some and how to call functions. And as we're going through today's code, I want you to sort of think about functions with two different hats on.

The first hat is from someone who's writing the function. So in the projector example, someone had to build the first projector. Someone had to know how to put all these components together. So that's going to be you writing a function, so you need to know how to make the function work. And then the other hat is you as someone-- as a programmer-- who is just using the function. You're assuming it's already been implemented correctly, and now you're just using it to do something.

So these are some of the function characteristics and we'll see an example on the next slide. So a function's going to have a name. You have to call it something. It's going to have some parameters. These are the inputs to the function. You can have 0 inputs or as many as you'd like.

Function should have a docstring. This is how you achieve abstraction. So it's optional, but highly recommended, and this is how you tell other people how to use your function. Function has a body, which is the meat and potatoes of the function-- what it does. And a function's going to return something. It computes its thing and then it gives back-- spits back some answer.

OK here's an example of a function definition and a function call. Function definition is up here. I'll just draw it here. This is the function definition up here. And this is the function call down here.

So remember, someone has to write the function that does something to begin with. So this is how you write the function. The first is whoops-- the first is going to be this def keyword. And def stands for-- it tells Python I'm going to define a function. Next is the name of the function. In this case, I'm calling the function is_even. And the function name should really be something descriptive.

Whereas someone who is just using this function or looking at it can pretty much tell what it's supposed to do without going a lot farther than that. They're just looking at the name. And then in parentheses you give it any parameters, also known as arguments. And these parameters are the inputs to the function. And then you do colon.

OK so this is the first line of the function definition. And after this, everything that's going to be part of the function is going to be indented. The next part is going to be the docstring, or the specification, and this is how we achieve abstraction using functions.

Specification, or the docstring, starts with triple quotes and ends with triple quotes, and you can sort of think about this as a multi-line comment. It's just going to be text that's going to be visible to whoever uses the function, and it should tell them the following things: What are the inputs to the function? What is the function supposed to do generally? And what is the function going to give back to whoever called it?

The next part is going to be the body of the function. We'll talk about what's inside it in the next slide. And that's it. That's all for the function definition. def blah, blah, blah, indented, everything inside the function. So this is you writing the function definition.

Once the function definition's written, you can call the function. And that's this part down here. And here, when you call function, you just say its name, and then you give it parameters. And you give it as many parameters as the function is expecting-- in this case, only one parameter.

So what's inside the function body? You can put anything inside the function body. You remember, think of a function as sort of a small procedure or a little mini-program that does something. So you can do anything inside the function that you can do in the regular program-- print things, do mathematical operations, and so on.

The last line is the most important part of the function though. And it's this return statement-- that's what we call it. So it's a line of code that starts with return, which is a keyword. And then it's going to be some value. Notice this is an expression here-- i%2 == 0 is an expression that's going to evaluate to some value. And as long as this part is something that evaluates some value, it can be anything you want.

And this line here return something tells Python, OK after you have finished executing everything inside the function, what value should I return? And whoever called the function is going to get back that value, and the function call itself will be replaced by that value.

OK so let's look at an example. I'm going to introduce the idea of scope now. And scope just means-- is another word for environment. So if I told you that you could think of functions as little mini-programs, the scope of a function is going to be a completely separate environment than the environment of the main program.

So as soon as you make a function call, behind the scenes what Python says is, OK I'm in the main program but I see a function call. I'm going to step out of this main program. I'm going to go off into this new environment. I'm going to create entirely new set of variables that just exist within this environment. I'm going to do some computations. When I see the return, I'm going to take this one return value. I'm going to exit that environment, and then I'm going to come back to the main program.

So as you're entering from one scope to another, you're sort of passing these values back and forth. So when you're entering a scope, you're passing a variable back into the function. And when the function's finished, you're passing a value back to whoever called it.

So once again, this top part is the function definition. And any arguments for the function definition are called formal parameters. And they're called formal parameters because notice they don't actually have a value yet. In the function definition, you're sort of writing the function assuming that, in this case, x is going to have some value. But you don't know what it is yet. You only know what value x takes when you make a function call down here.

So this is your function definition, and then later on in your main program, you might define some variable x is equal to 3. And then you make a function call. f of x here is your function call. And it says, OK I'm calling f with the value 3, because x takes the value 3, and then I'm going to map 3 into the function. The values that are passed into the function call are called actual parameters, because they're going actually have a value.

So let's step through this program-- this small program-- and see what exactly happens behind the scenes in the scope. And if you're just starting to program, I think it would be highly valuable if you take a piece of paper as you're doing some of these exercises and you write down something similar to what I'm going to go through here. I think it'll help a lot, and you'll be able to see exactly step-by-step what variables take what values and which scope you're in.

So here we go. When the program first starts, we're creating this global scope. It's the main program scope. In the main program scope, the first thing that Python is going to see is this part here-- def f of x and then some stuff inside. This tells Python I have a function named x, but I don't care what's inside the code yet. I don't care what's inside the function definition yet, because I haven't called the function yet.

So to Python it's just some code just sitting in the global scope. So whenever you see def, you're just putting some code in there. Then you go onto the next line-- x is equal to 3. So in the global scope, you now have also a variable x is 3. And then the next line-- z is equal to f of x is a function call. As soon as you hit a function call, you create a new scope-- a new environment.

So we're temporarily leaving the global scope and sort of portaling into a new scope, where we're going to try to figure out what this function's going to do and what it's going to return. So the first thing you do is you map the parameters. So x here-- I'm calling f of x with 3-- so first thing I'm doing is I'm mapping every one of the parameters in the definition to their values. So first thing I'm doing is x gets the value 3.

Next line here is x is equal to x plus 1. So we're still inside the function call f, so x gets the value 4. We're printing this and then we're returning x. So in the scope of f, x is equal to 4, so we're returning that value back to whoever called it, which was this function call within the global scope. So this part right here-- f of x, which was the function call-- gets replaced with 4. So inside the main program, z is equal to 4.

And that's how we pass parameters into the function, and we got a parameter back from the function. As soon as the function returns something, the scope that you were in for the function gets erased. You forget about every variable that was created in there, delete that scope, and you're back to wherever you started calling it.

One warning though. So what happens if there's no return statement? I said that every function has to return something. If you don't explicitly put a return statement, Python is going to add one for you. You don't have to do this. And it's going to actually have return None-- N-o-n-e. And None is the special type-- None is the value for a special type called NoneType, and it represents the absence of a value. What's that?

Not a string.

Not a--

None is not a string.

None is not a string, exactly. It's a special type.

OK so before we go on, I wanted to go through a small exercise in Spyder just to show you the difference that None and printing and returning makes. So here are two functions that I wrote. One is is_even_with_return. That's its name, so pretty descriptive. It's pretty much the same code we saw in the slides. It just has this extra little print thing. It gets the remainder when i is divided by 2. And it returns whether the remainder is equal to 0. So it'll either return a true or a false-- a Boolean.

OK so my function call is this: I'm saying is_even_with_return with a value 3. When I make this function call, this 3 gets mapped into here-- this variable here-- so i is equal to 3. I'm going to print with return, and then I'm going to say remainder is equal to 3 percent 2, which comes out to value 1, because there's a remainder 1. And I'm going to return whether 1 is equal to 0, which is false.

So this line here returns false, but am I doing anything with the false? Not really. It's just sort of sitting in the code here. So this gets evaluated to false. I'm not printing it. I'm not doing any operations with it. It's just sitting there. So it won't show up anywhere. If I want the result to show up somewhere, then I have to print it. So that's what this next line is doing. So that one should be straightforward.

is_even_without_return's a little bit trickier, but not too bad. I have print, without_return inside here, and then I'm going to get a remainder is equal to i percent 2. And notice that I'm not-- I don't have any return. So implicitly, Python's going to add a return None for me, like that. You don't have to add it.

So when I make the function call here, it's going to do the same thing, except that return in this case is not going to be a Boolean. It's going to be this special None. So this is going to get evaluated to None.

Again I'm not printing it out. It's just sitting there. If I were to print out the result of that, you'd be printing out this value None, which if I run it, you'll see here it just prints it out right there.

So as you're doing your next p set, it's about functions and you're seeing these Nones popping out in some places. Check to make sure that you've actually returned something, as opposed to just printed something inside the function like we did here. All right so that's the difference.

And the last thing I want to mention about this is_even function is how useful it can be. So notice this is the function as in the slides, and once you write the function once, you can use it many, many times in your code. So here I'm using the function is_even to print the numbers between 0 and 19, including and whether the number is even or odd.

So notice this piece of code here, once I've written this function is_even, looks really, really nice right? I have for all the numbers in this range if the number i is even, this is going to return a true or false for all the numbers 0, 1, 2, 3, 4. If it's true, then I'm going to print out even, and otherwise I'm going to print out odd.

So if I run this, it's going to do this. 0 even, 1 odd, 2 even, and so on. So notice using functions makes my code really nice looking. If I wasn't using functions, I'd have to put these two lines somewhere inside here and it would look a little bit messier.

So I've said this maybe once or twice before: in Python everything is an object. Might not have meant anything back then, but I think you're going to see what I mean using this particular example. So if in Python everything's an object-- integers are objects, floats are objects, even functions are objects. So as you can pass objects as parameters back and forth as function parameters, you can also pass other functions as parameters.

Let's see what this means. So we have three function definitions here-- func_a, func_b, and func_c. And then I have three lines of code here in my main program. So I have one called a func_a, one called a func_b, and one call to func_c. Let's trace through, just like in the previous example, and see what exactly happens.

First thing I create is my global scope. And I have three function definitions. Again I don't care what's in the code yet, because I haven't called the functions yet. Python just knows there's these functions with these names that contain some code.

After these definitions, I come to this line here-- print func_a. As soon as I make a function call, I'm going to create a new scope and I'm going to hop into there. Inside func_a, I'm going to go and look at what func_a does. It doesn't take in the parameters, it just prints out this message here. And then it leaves; it's done. There's no return, so we return None. So func_a returns None to whoever called it, which was that line there, so that is going to be None.

Next line. This one right here-- print 5 plus some function call. Again I'm going to hop into func_b's scope and see what to do there. So first I'm going to map my parameters. So 2-- whoops-- 2 gets mapped to y. So inside func_b's scope, y is going to get the value 2.

That's the very first thing I'm doing-- mapping all the parameters. Then I'm going to print this thing here, and then I'm going to return y. So inside func_b, y has the value 2, and I'm returning 2 back to whoever called me. So this is the value 2 and I'm going to print 5 plus 2, which is 7.

Last one. This is the trickiest. Oop, that popped up. If you think you've got it, try that exercise. But otherwise follow along. print func_c func_a. So I see that I am going to enter func_c's scope. So I'm going to look at what func_c does.

First thing I do is I'm mapping all the parameters. Don't even worry about the fact that this is a function right now. Just pretend it's x or something. So you say func_a is going to get mapped to the variable z inside func_c. So z is func_c. Just mapping parameters from actual to formal.

Then what do we do inside func_c? We print out inside func_c, and then we return z. This is the cool part. Inside func_c, z is func_a. So if you replace z with func_a, this here becomes return func_a open close parentheses. Look familiar? We did that function call right there right? So that's just another function call.

So with that being another function call, you're going to create another scope, and you're going to pop into that one. So we're one, two, I guess two scopes deep, and we're trying to figure out where we're going. So func_a's scope is going to be up here.

So what does func_a do? It just prints out this, and it returns None. So we're going to return None to whoever called us, which was func_c. So this line here becomes return None. And so this line here is going to return None to whoever called it, which was this line down here. Oops, I didn't mean to cross that out. So that line here is going to print None.

So if you just go step-by-step, it shouldn't be too bad to try to map what happens with variable names and formal parameters and actual parameters. That's why I highly recommend pieces of paper and pens.

One last thing I want to mention about scope before we do another example. So there are three sort of situations you might find yourself in. The first one is probably the most typical, and this is when you define a function. And it's using a variable named x in this case that's also defined outside of the function. And that doesn't matter because of the idea of scopes.

So inside the global scope, you can have variables x. When you're inside a different scope, you can have whatever variable names you want. And when you're inside that scope, Python's going to use those variable names, so they don't interfere with each other at all. So in this example, I've defined a variable x is equal to 1, and then I incremented, and that doesn't interfere with the fact that we have a variable x outside.

This one's a little bit trickier. I define this function g, and all g does is access a variable x. But notice inside g, I've never actually declared or initialized a variable x. In this f, I said x is equal to 1. But in here, I'm just sort of using x. So this does not give you an error. In fact it's OK for you to do this in Python.

Python says, OK I'm in this scope, but I don't have a variable named x, so let me just go into the scope of whoever called me. So I'm going to just temporarily hop out of the scope and see is there variable x outside of me? And it'll find this variable x here, and it's going to print out its values. So that's OK.

This last example here is actually not allowed in Python-- similar to this one-- except that I'm trying to increment a value of x, but then I'm also trying to reassign it to the same value of x.

The problem with that is I never actually initialized x inside h. So if I said-- if inside h, I said x is equal to 1, and then I did x plus equals to 1, then it would be this example here-- f of y. But I didn't do that. I just tried to access x and then incremented and then tried to reassign it. And that's actually not allowed in Python.

There is a way around it using global variables. But it's actually frowned upon to use global variables, though global variables are part of the readings for this lecture. And the reason why it's not a great idea to use global variables is because global variables sort of give you this loophole around scopes, so it allows you to write code that can become very messy.

So using global variables, you can be inside a function and then modify a variable that's defined outside of your function. And that sort of defeats the purpose of functions and using them in writing these coherent modules that are separate. That said, it might sometimes be useful to use global variables, as you'll see in a couple lectures from now.

OK cool. So let's go on to the last scope example. OK this slide is here, and notice I've bolded, underlined, and italicized the Python Tutor, because I find it extremely helpful. So the Python Tutor-- as I've mentioned in one of the assignments-- it was actually developed by a grad student here, or post-grad student slash post-doc here.

And it allows you to go through Python, paste a code, go through it step-by-step. Like with each iteration, it'll show you exactly what values each variable has, what scope you're in, when scopes get created, when scopes get destroyed, variables within each scope. So pretty much every single detail you need to sort of understand functions.

As we're starting to-- you can see we've had couple questions, and these were great questions. So if you're still trying to understand what's going on, I would highly suggest you take a piece of code and just run it in the Python Tutor and you should be able to see exactly what happens, in sort of a similar way that I've drawn my diagrams.

In all of the codes for this particular lecture, I've put links to the Python Tutor for each one of those exercises. So you can just copy and paste those, and it'll automatically populate it with that particular example, so you just have to click, step, step, step. OK so having made my plug for Python Tutor, let's go on.

OK so here's an example. It's going to show couple things. One is print versus return, and also this idea of you can nest functions. So just like you could have nested loops, nested conditionals-- you can also nest functions within functions. So let's draw some diagrams just like before of the scopes.

First thing we're going to do is when we have a program, we're going to create the global scope and we're going to add every variable that we have. And then when we reach a function call, we're going to do something about that. So the first thing in the global scope is this function definition. Again in my global scope, I just have g as some code because I have not called it yet. I only go inside a function when I make a function call.

So g contains some code. So we're done with 75% of that code. Next line is x is equal to 3. So I'm making x be a variable inside my global scope with value 3. And then I have this z is equal to g of x. This is a function call. When I see a function call, I'm going to create a new scope. So here is the scope of g.

With the scope of g, I'm mapping variables to actual parameters to formal parameters. So the first thing I'm doing is I'm saying inside g what is the value of actual parameter x? And x is going to be the value 3, because I've called g of x with x is equal to 3.

Next, what I see inside this function-- so this is the inside of the function-- is this bit here. It's another function definition. Again since I'm just defining the function and I'm not calling it, all Python sees is h is some code. I haven't called the function h yet, because I'm just defining it here with def. So that finishes this part here.

The next line is x is equal to x plus 1. So inside the scope of g, I'm incrementing x to be 4. Then I'm printing out this line. And then I've reached here-- h. This is actually a function call, and I'm calling h. As soon as I make a function call, I'm creating another scope. So I'm temporarily going out of the scope of g and going into the scope of h.

So Python knows that h contains some code, and now I can go inside h and do whatever I need to do. So the first-- so h doesn't have any parameters, so I don't need to populate anything like that in there. h does define a variable called x, which is abc; it's a string. And then that's all h does. What does it return? None.

I heard murmuring, but I think None was what you guys were saying. So since there's no return statement, h is going to return None. So h returns None. Back to whoever called it, which was this code inside g. So that gets replaced with None-- the thing that I've-- this circled red h here. As soon as h returns, we're going to get rid of that scope-- all the variables created within it-- and we're done with h.

So now we're back into g. And we just finished executing this and this got replaced with None. We're not printing it out, so this doesn't show up anywhere; it's just there. So we're finished with that line.

And the next line is return x. So x inside g is 4, so 4 gets returned back to whoever called it, which was in the global scope here. So this gets replaced with 4. So once we've returned x, we've completely exited out of the scope of g, and we've come back to whoever called us, which was global scope and we've replaced z is equal to g of x and that completely got replaced with 4-- the returned value.

So that's sort of showing nested functions. All right just circling back to decomposition-abstraction. This is the last slide. You can see if you look at the code associated with today's lecture, there are some other examples where you can see just how powerful it is to use functions. And you can write really clean and simple code if you define your own functions and then just use them later.

And the beauty of defining your own functions that you can use multiple times later is you only have to debug the function once right? I know debugging is not your favorite thing, but you only have to debug this one thing once, and then you can know that it's right and it works well, and you can just use it multiple times. All right thanks everyone.