[SQUEAKING]

[RUSTLING]

[CLICKING]

ANA BELL: OK. So last lecture, we started talking about the idea of decomposition and abstraction. And we talked a little bit about what that means and how it ties into what we've already been doing. Today, we're going to do a real-world example of decomposition and abstraction, and then we'll see exactly how we can achieve this in programming.

So let's start by talking about an example in the real world, the smartphone. So a lot of us have it, but for a lot of us, it's really just a black box. For me, I know it is. For most of the people in the world, the phone is a black box.

We basically view the phone in terms of its inputs and in terms of its outputs. So the phone has some buttons. You can scroll. You can touch things. But we don't really know exactly how all of these buttons and scrolling and all these internal workings actually do their job.

And in fact, we don't need to know how they do their job. To us, as the user, all we really care about is the interface between us and what task we want to achieve. So what we need to know, that interface, is basically the relationship between the input we give to the phone and the output we get. So when we push that button, the phone turns off. When we push this other button, the volume increases.

And so that's the idea of abstraction. The phone basically abstracted away all of those hardware details, all of those implementations that make it actually work for the user. So the user doesn't need to know how it works in order to use it.

Now, abstraction actually enables the decomposition. What does that mean? Well, once we abstract away details, we can have different manufacturers working on different components of the phone to build these different components. And if different manufacturers are working to build these hundreds of distinct parts within the phone separately, they need to have some way to put these parts back together.

And when they're working on their pieces separately, that's the idea of decomposition. How do they know that what they're working on will actually fit in with the rest of the components? Well, they use the idea of decomposition. They're basically following a specification. They're following a set of inputs that maybe come into their component, and a set of outputs that maybe their component needs to give to other components.

And all these different manufacturers that are building these different parts need to know is that interface bit. They don't need to know how other manufacturers build their components. All they need to know is what functionality those other components have.

And so all of these different manufacturers can build all these different components. The interfaces are going to be standardized, so to speak. And that's all that they care about. So once you know the interface, you can come together and put all these different components together to work towards a common goal, as in, to make a phone work.

So this is true for hardware, as in the phone example, but it's also true for software. And that's exactly what we will be doing in this lecture on functions. We're going to achieve decomposition and abstraction in programming, so treating code as a black box and making a large program, kind of splitting it up into these different self-contained parts.

So in programming, we want to suppress details as well, not just in hardware, like with the phone. We want to suppress details in programming as well. And we do this using this idea of abstraction. So we will be writing code, as we have already been doing, with the thought that the code we're writing will be done-- will be written only once. We will have some functional piece of code that will do a very useful task, and then, after we've written that code and debugged it and made sure it works well, we'll treat that code as a black box.

So from there on out, as long as we know what inputs that piece of code needs and what outputs that piece of code gives back to somebody else or to us, we don't care exactly how it does it. We just care that it is there, and it is available for use. So today's lecture, we're going to be seeing how we can actually create these little functional pieces of code.

We can then give these pieces of code to ourselves. We can definitely use these functional pieces of code that we've written, or we can give them to other people so that they can use them as well. So we're going to write these functional pieces of code. And we'll call the-- we'll call them functions or procedures.

And in fact, we've already been using functions, believe it or not. These three are examples of functions we've already been using in Python. So max is a function. It's some useful piece of code that, when we use it in this particular way, it says it's taking in two inputs and it gives me back the biggest of those two inputs.

The middle one, abs, is the absolute value function. And its input is one number, an integer, and it gives back to me the absolute value of that number. And len is also a really useful one that we've been using with strings. And basically, its input is a string, and its output is going to be how many characters are in the string.

So we've already been using functions. The point of today's lecture is you're going to start to write your own functions. Hopefully, useful ones. So the idea of a function is that we want to abstract away exactly how that function achieves something useful, some useful task.

And so the way that we're going to tell other people how to use our function is using this idea of abstraction. And we capture what the function does with these things called specifications. They're also called docstrings.

And the docstring is kind of like a contract between somebody who creates this useful function and somebody who wants to use the function. The person who uses the function might be you, the person who wrote it, or it might be somebody else. And in the contract, the things that we're going to mention are, what are the inputs to the function? So in the length function, it needs a string.

What is the function doing? And what is the output of the function? What does the function give back to somebody who uses this function? And we haven't actually done this explicitly, but you've probably seen this as you type your code in.

So here's an example of the absolute value function. And it comes up with this little pop up here whenever you type it in. So for example, abs, parentheses, right here, or if you hover over a function in your file editor, you'll see it pop up this little text box that says the specification or the docstring.

And here, you see exactly the signature of the function. So it takes in one input, the x, the value you want to find the absolute value of, and then, some text here, which is what the function does. So the specification of the docstring is literally just a multi-line comment. There's nothing special about it. As long as you hit those points, the inputs, what the function does, and what the function gives back to you, you've written a good specification.

Also, I should mention that these contracts, even though I call them contracts, they're not actually enforced by Python. So it's really just up to the person who writes the code to make sure that their specification is really detailed and your function does what you say you will. So if your function can take in both positive and negative integers, for example, then you better make sure that the function itself, doing whatever task it needs to do, can handle both positive and negative integers.

So once we write these functions, we now have these little bits of code that perform some useful task. Given some input, it'll perform this task and give me back some output. And now that we have these different little pieces of functionality, we can go ahead and take this large file of code, which you might write from now on, and see exactly which pieces of code maybe are getting repeated.

That's a clue that that's something that you can abstract away into a little module, aka, a function. And then, you can just use these functions to break up the code, a very large piece of code into smaller, little self-contained modules. And then, maybe the bulk of the work that the code is doing is just saying, hey, this module, please give me this answer. And then, this module, give me this answer, and then putting those values back together again.

So these reusable pieces of code are called functions or procedures. We're basically going to try to capture some sort of computation, a very useful task that you'd want to do over and over again. And we're going to see some details about how to write functions, but essentially, it's just going to be code that you've already been writing, just written in a special way that makes it reusable. So we're going to have to switch the way we've been thinking about code for a bit now that we're introducing functions.

Because right now, when we've been writing functions in a file, we basically write some code top to bottom, and then, we think about that code as being executed line by line, top to bottom. But now that we're creating these things called functions, little blocks of code that we can use many times in many different places in our code, we're actually going to introduce the idea of defining a function. So that means we're going to write a piece of code, and all that piece of code is going to do is tell Python that this is a module or a function that exists in my program. All we're doing is defining the function.

But we're not actually going to run the function when we define it. And that's the difference, the way we're going to have to shift our thinking here. So when you define a function, you just tell Python that here is some useful piece of code that exists that does something, but it doesn't actually run until you call the function.

And the cool thing about writing a function is, once you wrote it once, you can make 100 different function calls to that one piece of code that you wrote later on in your program. So you can call the function many times with different inputs to give you different outputs, but you only had to write it one time. So I would compare this to when we write code in a file.

When we write code in a file, yes, we can write a whole bunch of lines, but this code isn't running as we're writing it. We have to actually push the Run button to run that file. So similarly, when we're telling Python that I'm going to create this function that does something useful, it's not actually running those lines. We have to tell Python that we want to run this function.

So the first thing we're going to do in this lecture is just actually create a function. I'm going to show you how to define a function, so tell Python that this function exists. And then, we'll see how to actually run the function to give us some useful values.

So the function characteristics are going to be, the function has to have a name. So just like when you create variables that store some useful value, like pi to 20 decimal digits that you want to reuse over and over again, we're going to create a function, and that name is kind of like a handle for us to refer to this large chunk of code that does something useful for us. A function has some inputs called parameters or arguments. It can have no inputs, or more, or one or more.

And a function should have a docstring. So this is the specification. Again, just a multi-line comment that tells the user, the person who wants to use this function, the inputs, what the function does, and what the output, or the return, or whatever this function will do for you.

And then, the body of the function is just Python code, exactly the kind of code we've already been writing, except not wrapped in a function. So if you found yourself writing a piece of code that did something useful, you can totally wrap that in a function. And we'll see how to do that today.

So the body of the function is just a bunch of lines of code that do this useful task. The only difference in the body is that, at some point, this function has to end. It has finished its task. It figured out a final value, this useful thing that's the result of your task, and you want to give this value back to somebody who's using this function.

And we do that using this return keyword, as we're going to see in the next slide. So here's an example of a really simple function. So it's just a definition. So again, these lines of code do not run. Here, we're just telling Python that we're creating a function that does something.

So we kick that off with the def, define keyword. So notice it's blue. If you type it in your code, you'll notice it changes color. So def tells Python we're defining a function.

The next is the name of the function. So this should be something descriptive. Usually, it's an action word, because a function does something. So you want like, an actiony-type name for your function.

Then, we have parentheses. And inside the parentheses, you have any of the inputs, the parameters, the arguments to the function. So what should the user give you as input to this function? And then, of course, a colon.

So in that line right there, the only thing that is "customizable," quote unquote, is the name of the function and the parameters. If there's zero parameters you put nothing in there. If there's more than one, you separate them by commas. Everything else should be standard, the def, the parentheses, and the colon at the end.

Since we have a colon at the end, that means we have to indent the next bit of code. The indentation will tell Python that the rest of this is part of the function. So everything from here on out is part of the function definition. So we have our docstring, of course.

You start with triple quotes and you end it with triple quotes. And in it, you can write whatever you want. Just treat it like a comment that's on multiple lines. And you can see here, I've said that this function takes in an input, i, which I restrict to be a positive integer.

And then, I say what the input gives back to the user. So it will return true if i is an even number, and it will return false otherwise. So I've hit all the points, the inputs, what the function does, and what it gives back to whoever wants this function to run.

Beyond that, we have the body of the function. So here, you notice it's just lines of code that you would have written otherwise. If you were given the problem on a quiz that said, given i defined for you, write some code that prints true if the number is even and false if the number is odd. This is basically lines of code that you would type in.

The only difference is this little return here. The function is some lines of code that do a task. And that task, when it finishes, has to give something back. It can't just sit there, I guess.

And the thing that it gives back to whoever wants this function to run is set up by this return statement here. So if the number is divisible by 0, we return true, and else, we return false. So one of these either true or false values will be returned by the function. So this is you can think of it like the output of the function.

OK, questions so far? Does this make sense? Again, this is just us creating this function inside the computer, inside Python. We're not actually running these lines of code yet.

So if you are given a larger problem, I just want to take a couple slides to talk about how you think about writing the function. This was a really easy one. So obviously, it's not that hard to write, but what is the thought process if you are given a larger problem, like in English or something like that, and you wanted to translate this into a piece of code that does something functionally interesting?

So you think about what the problem is. So our problem is, given an integer, figure out if it's even or odd. So given this statement, you could come up with a name of this piece of code that's functionally interesting. So is_even is a good name.

And we can also come up with the inputs for this function. So i, we are only given one number, so there's no need for this function to take in any other inputs. And then, using that description, we can now start to fill in the docstring that says, well, our input is going to be a positive integer. We could use math to figure out restrictions on the inputs. And then, we can write the rest of the docstring that tells us what to return and when, what the function is doing.

And once you have that, you can just solve the problem. So for us, we solve the problem by saying, if the remainder when we divide i by 2 is 0, we return true, and otherwise, we return false. So that's code that you could have already written without, actually, this function lecture. But now, we're putting it in the context of a function definition. So we're going to be able to run this function with many different inputs to give us a bunch of different outputs, whether a bunch of these different numbers are even or not.

So when we're writing the body of the code, the only difference from what you've been doing is the return statement. Instead of printing something out to the console, we're going to return a value to somebody who wants to know whether the number i is even or odd. The function can also print stuff to the console. But the key thing here is you want to return a value to the user.

And after you wrote code, right off the bat, and you tested it and made sure it works, you can improve the code a little bit. So here, we're improving it by noticing that i%2==0 here is actually already a boolean. If i is even, 3%2==0 is true. And otherwise, it's already false.

So these four lines of code basically say, if true, return true. Else, return false. So our improvement can just be to return whether i%2==0 right off the bat. So here, we're going to return either true or return false based on what i is.

So at this point-- again, sorry I'm stressing this enough-- too much. But it's really important to understand that once we write these lines of code in the context of a function definition, these lines of code do not run. They basically just sit in Python saying that there are these lines of code that correspond to some function object whose name is even. That's it.

So what we need to do now is to actually tell Python to run these lines of code. To do that, we make a function call. And again, we've already been doing function calls, just to functions that already exist in Python, just Python itself, max, absolute, len, all that stuff. But now, we're making a function call to something that we wrote, this nice piece of code that tells us if a number, the input, is even or not.

So here, I'm going to invoke the name of my function, aka, I'm going to call the name of my function. I'm basically just typing in the name of my function in the code. Parentheses, and then, the inputs the function expects. There's only one, the number I want to figure out if it's even or odd. And then, that's it.

So I've got the name of my function, and then all the inputs, the parameters that this function expects. At this point, Python goes into the function body, it runs the function, and it returns back a value. So whatever the value is associated with the return is, that value will immediately be given back to whoever called it. What does that mean?

Well, that return value will completely replace this function call. Let's think back to expressions. Do you remember when we were learning about Python expressions and I said you have something like object, operator, object, like 3 plus 2? That was an expression.

And Python went in, evaluated that expression, and replaced that entire expression by the value, 5. This is exactly the same thing. In fact, functions are kind of like Python expressions. They do something useful. It's just that it's not math or something like that that gets evaluated. It's a bunch of lines of code that get evaluated.

But in the end, that function returns back only one value. And that value replaces the entire function call. So this entire function call is going to be, basically, replaced by false, because it's an odd number. And the next one is going to be replaced by true, the return from the function.

So the way that the code looks, just this definition of is even and then running a function call is this. This is all that we would have in our file. So here, we have our function definition, and then, at the same indentation level, we have a function call. Because the call is not part of the function. The call is just making use of the function that we wrote.

So what exactly happens? We'll do a little bit of step-by-step now going a little bit into more detail as to what exactly happens when we make the function call. So when we make the function call-- again, function definition. This just tells Python we have this function that does something in our program, and then, here, we have the function call.

As soon as Python sees the function call, that's when it starts doing something useful. Up here, it just sort of stores this in memory. So as soon as it sees the function call is even 3, it looks at the input parameter to the function call. And here, you see we have a value. It's an actual, tangible object. It's not some random variable. It's not something abstract. It's a number 3.

The i up here from our function definition is called a formal parameter. It's abstract. We wrote the body of the function assuming the user will eventually give us a value for i. But in the actual body of the function, i is just a variable we're using.

Kind of like in the quizzes. For now, I've been saying, assume you're given some number n that's defined for you. Write the code assuming this number. It's the exact same thing. We write the code at the body of the function assuming we know a value for i.

So when Python sees this function call with 3, it goes into the body of the function and says, all right, what are my parameters? There's only one. It's i. And it's going to map them one by one to all the actual parameters given in the function call. So basically just maps i to 3.

And then, it executes the body of the function. So it replaces everywhere you see i. So it might have a longer bit of code here, but here, we just have one line. It replaces i with 3. So we have 3%2==0. Now, we have a tangible value, false.

So this expression is replaced with false. And so this line of code here will return false. And as soon as Python sees that return value, it immediately exits the function and gives back the value that you're returning to whoever called it. So this entire function call here will be replaced by false.

That was very step-by-step, but does it make sense? OK. So this is a program that doesn't do anything. If somebody were to write this program and run it, it doesn't actually show anything to the user. That's because, in our program, it's like we had just written a line of code that said false. Does that get printed to the output? No, right?

What we need to do is do something useful now that we have the result of a function call. So one useful thing we can do is to actually print the result of the function call. So here, we have print, and then, I have my function call I had up here. I'm just sticking it inside the print statement.

And Python will, as before, evaluate is_even(3). This is replaced with false, and this line essentially becomes print false. And so the way this looks in our actual code is this.

So here, I have this is_even function, the inefficient way of writing it. I've got two function calls here. But if I run the code, it doesn't print anything. I need to do something useful with them. And one useful thing we can do is to print the result of these function calls. So now that I've wrapped these calls inside a print statement, I see the output in my console.

So we're writing-- so we're kind of separating ourselves when we're writing code now. One, we're defining a function, some code that does something useful. And then, two, we're using this function that we wrote to make function calls.

And the beauty about writing the function is we only write it once and debug it once, but now, we can run it as many times as we'd like. Without functions, we'd find ourselves copying and pasting that piece of code that does something useful in many places in our code, which could lead to errors. The code is hard to modify. It's hard to debug, all that stuff.

I'll give you a chance to try this out for about a minute. So let's have you write this code. So here, I'm giving you the function specification. Most of the time, I'll give it to you even in quizzes. I want you to write for me a function called div_by.

This one takes in two parameters, both integers greater than 0 and d. And this function will return true if d divides n evenly, and false if it does not divide n evenly. So if you test it out with those two values, the first one should give us false, and the second one should give us true.

So as usual, this is down in the Python file. Around line 28 is where you should start typing in your code. Does anyone have a start for me? Should be very similar to what we just-- yeah?

AUDIENCE: e%n==0, then print true.

ANA BELL: Else, print false. So let's run the function. Oh, let's just do it with one. So the first one, I'm expecting to print false. It does print false, but it also prints this weird none right after it.

Actually, this is something we want we're going to talk about next lecture. But does anyone know an improvement we can make to the code? Yes.

AUDIENCE: [INAUDIBLE]

ANA BELL: Yes, actually. You're right. So instead of printing true, remember, it's a function. We want it to give us back the value true, right? So instead of printing, we'll do a return true. And we don't need the parentheses in this case. And then, we'll do a return false.

So now, we don't have that weird none right after it. That's something I was going to talk about in next lecture. But basically, when we had print here, what did the function return? Did it have a return statement inside it? No, right?

And so if there's no return statement inside the function, Python automatically returns the special none. This is something we'll talk about in next lecture more in detail. But the return true, return false is correct here. Yes?

AUDIENCE: Do you need the details, or do you just return--

ANA BELL: Yeah. Yeah. You don't need the return. The if else, just like before. So we can just do return this directly. And we can run it with the other one. So the second one should actually return true. But it returned false. Does anyone know the problem? Yeah?

AUDIENCE: It has a remainder.

ANA BELL: Yes, exactly. So actually, we want the remainder when we divide n by d. So this is just flipped around and %d==0. Yeah. So it's a good thing we had two test cases to test for that. And you don't have to test them with such big numbers. You could obviously test them with some smaller numbers as well.

So let's zoom out a little bit and talk about how exactly functions are stored in memory. Because I mentioned this thing about defining a function, and that just doesn't do anything really that we can see. But what exactly happens in memory? Well, let's think about what happens when we create variables.

So when we create a is equal to 3 inside memory, or the program scope-- again, we'll talk about this next lecture. But you can think of this as the memory. What happens is a becomes a variable that's bound to value 3. b equals 4 is a variable b bound to value 4. And c is going to be bound to value 7. Clear, right? We already know this.

What happens when we create a function? So again, this is something I might write in a code file. The top bit is my function definition. So as soon as Python sees this def keyword, everything that's indented, that's part of the function definition in the body is essentially just some code.

To Python, it does not care at this point what that code is or what that code does. All it knows is that there is a function object-- and functions are actually objects in Python. There is a function object whose name is even. That is all it knows when we get to this point here in the code, right after we define the function, right before a equals.

So we think about the function as kind of like a "variable," quote unquote. It's not actually a variable, but it's like a variable whose name is even, and it points to-- it's bound to some code in memory. And we don't care what that code is right now because we might never use it. We only care what the code is when we make function calls.

So down here is where the action actually happens when we make our function calls. I have a is going to be, as usual, a variable. That's going to be bound to some value. So the function definition is kind of just like a black box. Once you wrote it once and it works, you don't care anymore how it actually achieves its task. All you care is that it takes in a number and tells you whether that number is even or odd via true false.

So down here where we make our function calls, we're just using our black box. And we're using the black box by making function calls. So a is going to be a variable that's bound to the value returned by is even. So it's going to be based on the function call, false.

And then, here, I have another function call. I'm using this useful piece of code that I wrote up here. And b is going to be a variable that's bound to true. And c is going to be a variable that's bound to true. Does that make sense? Kind of separating the code we write which doesn't run until we actually make function calls. That's the thing about functions. And that's how it helps us write more robust code.

So now, here, we can have a more complex piece of code where we're using the function that we wrote. Not just making a function call and printing the result, but we're actually using it inside a more interesting program. So here, I've got a program that will print for me the numbers between 1 and 10, and it'll print whether that number is odd or even.

So if you were just to read this code, it's pretty easy to read, right? You have a loop that goes through the numbers 1 to 10, not including 10, and then, I have this if is_even. Well, that's cool. Here, I'm using the function that I wrote kind of just in the middle of another piece of code.

Which is fine. Because as I said a few slides ago, function calls are basically just expressions. They get run. They get evaluated. You one value back out of them. And then, that value replaces the function call.

So that's fine. Let's use the is_even result, the return from the is_even method inside a conditional. If calling is_even with i returns true, that means if the number is even, we print that value, comma even. Else, we print that value, comma odd.

So here, I'm not defining a function. Notice, it's not wrapped in a def or anything like that. I'm just using a function that I already wrote. So inside here. I'll just comment that out.

This is the code we just had on the slide. So again, notice it's not within. It's not wrapped within a function. It's just a loop that tells me the numbers 1 at a time, whether they're odd or even. So prints 1, comma-- yeah?

AUDIENCE: What are you doing when you select [INAUDIBLE] comment and you make it like--

ANA BELL: Oh, when I select everything. I just use spider-like ability to-- so I do Control 1, or Command 1, probably, on a Mac. And it just comments and uncomments things in batch. Yeah. Very useful. Yeah.

And so this code is now very easy to modify. I can just choose 100 and then I can run it again. And it gives me the numbers 1 through hundreds odd or even. And you can imagine using your is_even function in a more complex setting.

And is_even is a really simple function too write, but again, you can imagine writing a more complex function. And then, that complex function isn't a whole chunk of code that just gets stuck into this program, this loop. It's going to be a function that you call that you can just easily read the specification for and you don't need to completely understand how it works in order to use it.

So we're going to go through one other example to write a little function. And this will also showcase the best practices for writing a function and writing code, especially maybe in a quiz situation or something like that. How to write incremental code, how to test it a little bit at a time, and so on. So the last example I want to go through is I want to write some code that adds all the odd integers between and including a and b.

Might be something you're asked on a quiz. The first thing you do when you're faced with such a task is to think about a nice name for the function. So sum_odd or sum_odds is a reasonable name.

The inputs to the function, well, I've got two endpoints. I want to sum odd numbers in between. So the inputs might well be my two endpoints. And then, what is the thing your function achieves?

Well, in the end, it's going to give me some sum. So let's call that sum a variable sum_of_odds, and we'll return it at the end of our function. And in between, we're going to have some code.

So first thing to do is to not write code right away. When you're faced with a task, again, on a quiz or something like that, it's best to take a piece of paper, write a little bit, one example, and try to think about how you'd solve it, not like a human would. Because for us, we would immediately know the sum. It's very easy for humans to identify the solutions to these problems.

But try to think about how you would write-- what kind of a recipe would work for this? Would you loop? Would you have a conditional? Would you use a for loop, or a while loop, and a bunch of other concepts that we'll learn about in the following lectures. But the key thing is to just not write code right away.

So if we start with a really simple example on paper, we can say, let's choose end points a is 2 and b is 4. On paper, I would probably write out 2, 3, 4 in a row. So I know the numbers I need to look at. I would say 2 is my a, 4 is my b. I need to look at every one of these numbers one at a time. Reasonable.

I can do another example. Sorry. And I know what the answer should be. So I figure out what the answer should be so that when I write my code, I actually know what I'm looking for.

I look at another example, let's say a little bit more complicated, a bigger range. a is 2, b is 7. I try to use the same strategy I used, same recipe I used to solve that simpler example in this harder one. So again, I'm going to write out all the numbers between 2 and 7 inclusive. This is my first, this is my last.

And my strategy was to go through one at a time, and if it's odd, I take it to my running sum, add it to my running sum, and if it's even, I don't. I ignore it. And again, I know the answer for this should be 50.

So with these two examples in mind, I can start writing code. But instead of writing code for the big problem that might include some nuances or some edge cases, I can actually try to solve a similar problem. So instead of summing all the odd numbers between a and b, let's just sum all the numbers between a and b and see if we can get code working for that. Once we do, figuring out the odd ones should be a small tweak to our code.

So if we start with figuring out the sum of all the odd numbers between and including a and b, that sounds like a loop. Because I knew, when I wrote my example on paper, I'd have to touch each number between and including a and b. So I know I need to loop through every one of these values.

While or a for loop? Your choice. In the slides, I'll do both, just to see what it looks like. So with a for loop, it's easy. It's just for i in range a b. But with a while loop, remember, we have to initialize our loop variable, if we have one. i equals a.

Our loop condition is while i is less than or equal to b. So we're going to loop through while I'm looking at all these values up to and including b. And I need to remember to increment my loop variable within the loop. By one each time, in this case.

And then, what do I do within my loop? Well, I'm going to-- remember, we're solving a similar problem. I'm going to keep a running sum. So as soon as I see a new i, I'm going to add it to my sum.

I realize here, probably my id would show me that there's an error. I didn't initialize sum of odd. So I remember to initialize sum of odds right before the loop. And then, this is a good place to test the code for a little bit. So we'll test it with a really simple example, 2, comma 4.

If we test it with 2, comma 4, the for loop gives me a 5, but the while loop gives me a 9. So you guys might have noticed what the problem is. My for loop goes through up to but not including the n variable, the b.

So we can add a print statement, in case you didn't figure that out, and the print statement would actually tell us. It tells us what we're incrementing. First, it's 2, then, it's 3, but I never hit the 4. So the fix is to just change my end range to be plus 1. And then, we run it again, and we see the answers match.

And this solves the bigger problem. So now, all we need to do is adding the nuance, the piece where we just grabbed the odd numbers. And here, we say, well, if I'm just grabbing the odd numbers, I only want to add i to my sum of odds when I see an odd number.

So here, I could use my is_even function that I already wrote. I would say if not is_even, or I can just do it all over again. If i%2==1, then we do this. And now, we can run it again, and hopefully, this now matches with the example I had on paper. And it does.

So the idea here is to try to solve a simpler problem first, and then, as you see more nuances to the problem, add in the functionality just a little bit at a time so you don't actually get bogged down by a whole bunch of issues that might come up when you wrote a whole bunch of code. The last step is just to test it on the other example, just to make sure that it still works. And so if we print sum_of_odds between 2 and 7 again, this matches what I had written down on paper.

If you don't want to use print statements, the Python tutor is also a great debugging tool. So testing code often, very useful. I think I've stressed this in previous lectures as well. Using print or the Python tutor to debug is also very useful. I don't actually intend to go through this try it, but this, along with a bunch of other examples, things to try at home are in the Python file.

So just functions you can write is palindrome, this keep consonants, this first_to_last_diff. Read the function specification and try to write code that matches the specification. And as usual, the answers are in the Python file but, please, please, try to do them on your own first before looking at the answers.

A quick summary. Functions are very useful. Allows us to abstract certain useful tasks. Basically, abstract away functionality that we might reuse many times in our program.

Functions take in inputs. They have something to return. We're going to see next time what happens when we don't return anything. Creating the function is different than running the function.

So you create the function once, but you can run it many, many times. And that's what makes functions useful. It makes code easy to write, read, debug, modify. Leads to a very nice robust code.