[SQUEAKING]

[RUSTLING]

[CLICKING]

ANA BELL: So today, we're going to wrap up talking about functions by talking about these things called lambda functions as a way for us to create anonymous functions. And that will pretty much finish our exploration into creating functions. And the last part of the lecture, we're going to introduce new object types, tuples and lists.

So let's remember what we did last time. We ended with this example. We created a function. You guys wrote it for me. And then we wrote it and debugged it together. But we created this function called apply.

So what was interesting about this function is that one of its parameters was a function and the other one was an integer. And that seemed a little strange at first, but not when we realized that functions in Python are actually just objects. And so they have a name, which means that anywhere where we use other kinds of objects, like integers, floats, we can use them as parameters to functions, we can use other functions as parameters to functions as well.

So here, criteria, we had just used it as a variable name assuming that the type of criteria is a function. According to this documentation, we assume that it takes in a number and returns a Boolean. So we just wrote the body of the function assuming that that is true. So right here, is where we used this function named criteria.

We assumed that it takes in an integer. So we passed in the loop variable I as an integer. And we assumed it returns a Boolean. So we were able to use the return of criteria(i) just as a Boolean inside as my condition for this if statement. So hopefully you got a chance to look through this example from last lecture.

So that's the definition of this function that takes in another function as a parameter. And then the way we use the function is down here. So "apply" is us making our function call. And then, the first parameter is the name of a function. And the second parameter is an integer. So the name of the function we're running is this object that we defined over here. Hopefully this is just review.

Now, what's interesting about this example is that this is_even function is pretty simple. It's basically a one-liner. It doesn't do any computations inside the function body. It basically just takes in a value, an input, and returns something.

And so we didn't really need to create a function, a full-fledged function definition just to do this really simple task. And in fact, that's what a Lambda function is. It's basically a way for us to create an anonymous function, a function that does something really simple, but we just don't give it a name.

And so here is the function that we created with an actual definition up here. We can create an equivalent anonymous function that looks like this. So this is a much more concise way for us to create a really simple function that we only need to use one time. So here is-- I'm going to just map out, one by one, the important pieces of the Lambda function.

So the Lambda keyword starts out the anonymous function. And it tells Python that we're creating this anonymous function. So lambda is not the name of the function, it just tells Python we're going to create this function in one line that is nameless. x is going to be any parameters that we expect this function to take.

So if we have more than one, we just separate them with commas. Colon is, again, the same. And then, the body of the function, if you can write it in one liner that's not too complicated, you can make a Lambda function out of it.

So here, notice, we don't actually have a return keyword when we're creating the Lambda function. We're just doing the operation that we wish to return the value from. So the x%2=0 is basically the body of my Lambda function over here. So the key thing about Lambda functions is that it allows you to create a really quick Function object that you basically want to use only one time. And so we're not giving it a name.

So let's look at the code. So here is my Apply function that we've seen before. Here is us-- I showed you this last time. I created another definition for another simple function that takes in an integer and returns a Boolean.

In this case, this function just tells me whether that input is equal to 5. And this is where we left off last time we ran apply with this is_5 function. So that prints apply with is_5 is 1. There's only one integer between 0 and 10 where applying this returns true.

Now, with an anonymous function, just to show you how we would write a Lambda function for this is_5, it would look like this. So again, we tell Python we're creating an anonymous function. It has just the one input x, colon, no return, and just the body of the function is going to be the thing that we would like to return, x==5.

So again, this notice, we're not actually passing in the name. There is no name for this anonymous function. But it works in the exact same way as if we had created this function over here. And I can run it again. And you can see "apply with" the function name is 1. And obviously, apply with this anonymous function also returns 1.

So just to bring the point home, I want to show you one other way to think of these anonymous functions. So here is me calling my is_even function with a parameter 8. Now, in order for me to actually run this line here, I had to have the function definition way up here. But again, it's a really simple function.

If I only want to use it one time, I can create a Lambda function. And this, over here, is equivalent to this function definition and a function definition over here. So you can think of this line over here, so the part that I've highlighted, as creating the definition all in one line, not giving it a name. And then, the parentheses here is us calling those lines of code for that function definition with that parameter 8.

And so the usefulness of Lambda functions is when you want to create these really quick functions that you don't want to reuse. Obviously, if we wanted to reuse the functionality of the is_even, but we created it using a Lambda function, we would have to basically copy this line and paste it all over again. So we'd have to take this, copy it, paste it, and give it another input because this Lambda function does not actually create it in memory with a name. There's no way for us to access the body because it's nameless.

OK, so just to finish how we call lambda functions, so basically when we called the apply (is_even , 10), the equivalent to calling that function name but with a Lambda function is basically putting in the entire body of the Lambda function inside this other function call. So here, we're both defining and then telling Python that this is my input to the function.

OK, so I know this is a You Try It. But I thought that we would actually run through it together step by step on the next few slides. So let's try to understand what this is doing. I've got a function definition named do_twice. It takes in one input, another input.

But if we look at the body here, this fn-- that's the input-- is actually being called a function inside the body. So we can immediately tell that fn is going to be a function when we actually make the call to do_twice. And indeed, when we make the call to do_twice down here, n is mapped to 3 and the second parameter, fn, is mapped to this anonymous Lambda function.

So let's step through one a little by little in the same manner that we learned last lecture, so creating actual environments whenever we see a Function call, mapping parameters-- actual parameters to formal parameters, and following through on what exactly happens within each function body.

So when we first make the Function call, right-- or sorry, when we first run this program, if it has these three lines of code inside it, Python creates our global environment. Inside the environment, we've got one Function definition here. So this is going to be this Function object. And then I've got the thing that actually kicks off my Function calls, my program.

So I've got a Print statement that will print the result of doing something. So the first thing I can see here is that I've got a function call to do_twice. So I'm going left to right.

The first thing I do when I have a Function call is I create a new environment. Inside this environment of do_twice, I have to see what it takes in. What are its formal parameters? There's one called n and one called fn. So there's one parameter n and the other one, fn.

And now, I basically map, one by one, the formal parameter to the actual parameter. So the n gets mapped to the 3 because that's the first parameter of do_twice. And the fn gets mapped to this Function object here. So the fn gets mapped to this Lambda function here.

OK, that's exactly what I said. So we've done the mapping. And now that we've done the mapping, we can do the body of do_twice. So the body of do_twice says "return." And then I have to replace everywhere I see "fn" with this Lambda function and everywhere I see "n" with this 3.

Well, fn is going to be a Function call. Whenever we see a function call, we need to create a Function scope. So before I can do the return, before this do_twice can terminate, can return its value, it sees a Function call. So when there's a Function call, we need to create another scope, another environment.

This environment belongs to the function call of lambda x colon x squared. Now, this function, of course, doesn't have a name. Normally I would say this is the f environment or this is the g environment or the is_even environment. But there's no name for this one. So I'm just going to write up here the body of that function.

All right, well, in this function, again, following the rules one by one, what we need to do is figure out what are the parameters of this function. Well, there's one called x. So here is my parameter x. And then I need to figure out, what does this map to. Well, what it maps to is the parameter inside it. But the parameter inside it is fn(n).

Do we have a return value for this yet? No, because this is another function call. So what ends up happening is this environment gets put on hold, as well, because we can't figure out what parameter this Lambda function takes in, what is its value.

So we create another scope, another environment. And in this particular case, this one is going to belong to this inside bit here fn(n). So this lambda x x squared is going to be the exact same function again, being called again. And in this particular environment, we need to map x to its input.

So the input to this lambda x x squared is going to be n. Well, this environment doesn't know about n. So we pop up one level. This environment knows about n. It's 3. So it passes that value along down to this Lambda call.

So now that this inner highlight yellow over here knows what it needs to do, it needs to take in this x and return x squared. So it calculates 9 and then returns 9 to whoever called it. That 9 gets replaced now as the input to this outer fn. So just to show you exactly what gets replaced, that entire Function call there gets replaced with 9.

All right, as soon as we've done the return, that environment goes away. And at this point, this call to lambda x x squared can terminate as well because it takes in the number 9 and it returns 9 squared. So this one returns 81.

So this entire Function call is 81. And as soon as it returns, that environment goes away. And now do_twice can finally finish its job and return 81. It just basically passes this value along back up. So that returns 81. So this entire do_twice call is going to be 81.

AUDIENCE: Why does lambda-- why were there two of them again?

ANA BELL: There were two of them because this outer fn calls an inner fn, so we--

AUDIENCE: Oh.

ANA BELL: Yeah, OK. OK, so that wraps up our discussion on functions. And there's a couple exercises in the Python file associated with this lecture with Lambda functions just so you can give it a try with those. Yeah, question.

AUDIENCE: With lambda functions, so we use print or apply, because if we [INAUDIBLE]?

ANA BELL: Well, apply was just a function that I wrote. So in this new example, I was just printing the result of calling that function.

AUDIENCE: OK.

ANA BELL: Yeah. So again, this kind of trace of what happens throughout the program is really, really useful. So if you have some time to try to get that down, it'll be very, very helpful as you trace through some programs.

OK, so that ends our discussion on functions. And really, the only syntax we've introduced in the past couple lectures were just about how to wrap code we've already been using in a function. So not much new syntax. But today, we're going to introduce some new syntax along with the introduction of two new data types. One is called a tuple. And the other one is called a list.

So what are the data types we've seen so far? We've seen integers, floats-- basically numbers. We've seen Booleans as truth values. We've seen this none type, type which has one value none. And we actually also saw the string data type.

We could think of the string data type as a compound data type, like a sequence of single characters. And in fact, we were using that string in that way because we were able to index into the string to grab the character at index 0, sort of slice the substring to get the length of the string.

Today, we're going to introduce two more compound data types, so these things called tuples and these things called lists. And throughout the lecture, you should really think about how it's very, very similar to the strings that we've already seen. So a lot of the operations, I'm actually going to skip.

Aside from the syntax of how we denote a tuple or a list, really, the operations that we do with tuples and lists are going to be exactly the same as the ones that we did with strings. So if you understand indexing and slicing and getting the length of the string, all that stuff, you'll understand how to do that for tuples and lists.

All right, so tuples are indexable ordered sequences of objects. That's kind of a lot. So we can break that down. So first of all, it's a sequence of objects, just like a string was a sequence of single characters. A tuple is going to be a sequence of not just characters but any kind of object.

Ordered sequence means that there will be an order to this sequence. So there's going to be an object at the first position in my tuple, an object at the second position in my tuple, and so on, just like there was a character at the first position, character at the second position, and so on. And indexable ordered sequence means that we can index into this object. So we can grab the element at index 0, grab the element index 1, and so on and so on.

So how do we create these tuples? I should note that some people call them "tup-ples" because they're just kind of like an n-tuple kind of thing. So you can call them "too-ples" or "tup-ples," however you'd like.

All right, so how do we create these tuple objects? Well, we can create a tuple object that's empty using just open and closed parentheses. So we could create strings using just the open and closed quotation marks. We create an empty tuple by doing open and closed parentheses.

Now, this is different than functions. This is a little bit similar. It might be a bit confusing because we use parentheses to make Function calls. But notice, it's just the parentheses by themselves. There's no function name, nothing preceding the parentheses. So to Python, it's not going to be confusing when you just do this.

You can create a tuple with one element in it by putting open closed parentheses, that element that you want to add to your tuple, and then a comma right after it. Now, the comma is there to differentiate a tuple with one element from using parentheses as precedence over an operation.

So just as an example, if I create a is equal to 5, like this, I'm using parentheses around an integer. But the type of a is still an integer. I'm basically just using the parentheses to say, I want to do this 5 before doing anything else, which is a little strange to do. And write the value of a is 5.

But if I do b is equal to the tuple 4 comma, this tells Python that this is now a sequence of objects, but there's just one object in my sequence. So the type of b is a tuple, not an integer. And if I ask what b is, you can see it's 4 comma in parentheses. It's a tuple with one object in it.

OK, so to create a tuple with many objects in it, we basically put in parentheses all the objects I want to add in my tuple separated by commas. So here, I've got my first element in my tuple, integer 2, second element in the tuple, the string mit, and the third element in my tuple being the integer 3.

And notice, we can mix and match now objects of different types within my tuple object. So here, I've got integers and strings. And integers, I can even add floats and Booleans. And whatever object types I'd like, I can make them elements to my tuple, which is pretty cool, right? Different than strings in that respect, but still in order within my tuple.

And so the rest of this is actually operations that we've already seen on strings. So I'm not going to go through them in too much detail. We can use the square bracket to index into the tuple, so to grab the element at a particular index.

Again, indexing starts from 0. We can use the plus operator to concatenate two tuples together to create one larger tuple with all those elements in a row. We can slice down here. We can get the length of the tuple, which tells us how many elements are in it, so three elements.

We can use the max, min, some sum things like that to grab the maximum element, minimum element, sum all the elements of my tuple and things like that. Notice that here I've got parentheses for the max function call and then another set of parentheses here to denote that I have one tuple object I'd like to grab the max of.

And then, the last bit here is something that we're going to see that's different with lists in next lecture, not today. But basically, you might think that once you create this tuple object in memory, that has 2, mit, 3 as its three elements in it, you can go into memory and modify one of the elements.

If I don't want the middle one to be a string, I want it to be a common integer, you might think that you should be able to change it. You can with lists, as we'll see in the next lecture. But you cannot do this with tuples. Just like once we created an integer 5 inside memory, we can't go into memory and tell Python to change this 5 to a 6. It's just not allowed.

Or once we created a string abc in memory, you can't go into memory and change the string. You can certainly create new objects that are based on this string. But you can't go in and modify that object once it's created. So once you've made your sequence of tuples, you cannot go in and change it. Yeah?

AUDIENCE: I have a question. So if you just rewrote t equals, and then [INAUDIBLE] different, it would be like an error?

ANA BELL: If you wrote t equals and then something different?

AUDIENCE: Yeah, [INAUDIBLE] you can't modify one thing.

ANA BELL: Yeah, that's a good question. So the variable t, so the name t and the object it's bound to are two different things. So the object it's bound to will still sit in memory. We're just going to lose the binding from it.

So that t initially points to this one. But then if you say t equals something else later on, this one still stays there. But that t is going to point to this new thing.

So the object itself is still in memory. We've just lost the binding to it. And that's something we did way back in the first early lectures where we kind of rebound variables, yeah. So yeah, it's the same idea.

One interesting thing that we can do now with tuples that we couldn't with strings is to have elements of a tuple be another tuple. And that's what this example is going to showcase. So here, I've got an integer 2 as my first element. My second element is the string a.

My third element is my integer 4. And my fourth element is a tuple object that just happens to have two elements inside it. But this tuple object that I'm referencing by seq, seq only has four elements in it. It just so happens that the last one is a tuple.

But I'm not going to dive further down to figure out if I have tuples that have subtuples that have subtuples and so on. Only top level I care about how many elements I have. And so when I print the length of seq, it's going to be 4 because I have 1, 2, 3, and then this last object is just one object that takes up one slot. It happens to have elements within it.

And so the rest of these are basically what we've seen with strings except for this one here. If we were to index into the last element here of seq, 1 comma 2, well, this is another tuple, right? So it should follow that I can then take that tuple and further index into it. And so that's what this line here is doing.

When we read an expression, we go left to right. So basically, seq at index 3 grabs for me the 1 comma 2. And then if I further index into 1 comma 2 at index 0, I'm going to grab the number 1. So I'm basically chaining all these indexing operations together.

And then this is, again, very similar to what we've seen from strings. So it's just slicing instead of indexing into the tuple. I'm not going to go through it today. But I encourage you to type them in and type in some other things as you might have done with strings.

One thing that I do want to mention is that we can iterate over a tuple just like we could iterate over a string. I don't mean over indices. But I mean over the elements directly. So when we iterated over a string directly, we were able to grab in our loop variable the characters at each index.

Similarly, we can iterate over a tuple to grab the elements at each index directly. So here, I've got for e in seq is going to make my Loop variable e take on each element of the tuple directly, not the index but each element. So as I'm looping through, e will first have a value 2, then it'll have a value a, then it'll have a value 4, and lastly, it'll have this value 1 comma 2.

And so if I just print that out directly, you'll see these values printed out. So very, very similar to some of the operations we've done with strings. The only difference is we just now have to be careful that our tuples can have elements that are other tuples or basically any object in Python.

So what do we use tuples for? Well, there was this one example we did way back at the beginning of 6.100L where we tried to swap variables. And we basically said that this way didn't work because we overwrote the variable.

We overwrote the variable, and then we weren't able to get back to the value that was overwritten. So our solution was to create this temporary variable to save the value before we overwrote it, then overwrite the variable, and then use the temporary variable to grab back that saved value.

Well, it turns out tuples actually allow us to do these three lines of code in one line of code here. So we can say x comma y equals y comma x. So this is an assignment. And it's allowed because the left-hand side is basically a set of variables in sequence. And the right-hand side gets evaluated first as we would an assignment statement.

So y gets the value 2 because that's what it is up here. And x gets the value 1. So y is 2, x is 1 over here on the right-hand side.

And then Python, one at a time, matches the values on the right to the values on the left separated by commas. So basically what we have here is x is equal to 2, y is equal to 1, and then the values have been rebound. So very, very, very useful-- very good use of tuples here.

Now, this idea can actually be taken one step further. And we can use tuples to return more than one value from a function. Now, I know in the past couple lectures I said, basically you can't return more than one thing from a function.

A function returns only one thing. As soon as it sees a return statement, it takes the value associated with that return and returns it back to whoever called it. But tuples are one object. They just so happen to have elements that can have different values. You can have a tuple with 10 elements in it. You can have a tuple with two elements in it.

Using a tuple, we can actually return one object, the tuple itself. It just so happens to have a whole bunch of values that my function might calculate. And so by way of the tuple, I'm actually able to return a whole bunch of different values through this one object tuple.

And so in this particular example, I have a function that calculates the quotient and the remainder when x is divided by y. Yeah, so the function itself uses integer division to find the quotient and uses the remainder operator to find the remainder, and then it returns that q calculation, some number, and that r calculation, another number, as elements to a tuple.

And Python returns this one tuple object using this line here returning this object. And so when I make this Function call to quotient and remainder 10 comma 3, it's going to go in, it's going to calculate the quotient to be 3, the remainder to be 1, and it's going to return one object, 3 comma 1. And then that gets assigned to this variable, that I named "both" in this particular case.

If I wanted to access the quotient part of both, I would do both square brackets 0. And the remainder part of both would be both square bracket 1, accessing the zeroth element and the first element of the return. Now, if I wanted to explicitly save the quotient and remainder as variables after they got returned, I can actually do the trick we saw on the previous slide. The trick that we saw on the previous slide was x comma y equals some other tuple.

Well, that's what I'm doing here. I'm making a function called a quotient and remainder 5 comma 2. That's going to return 2 comma 1. And then I'm going to have quote comma rem equals 2 comma 1. So Python, one at a time, is going to map the quote to 2 and the rem to 1.

And so what that means for us in terms of the code is we can then do whatever we'd like in the remaining part of the code, code assuming that quote and rem are just regular variables. So here, I'm just showing that you can print them out in these print statements. So here, I have quotient is 2 and remainder is 1 as these two lines of code here.

OK, so the big idea of tuples and the reason why we use them is you can use them to return more than one value via this one tuple object from a function. And so in this way, we can have a function that does a whole bunch of calculations, returns this one object that might contain all of these different values as the elements to this tuple object.

So let's have you work on this for a couple of minutes. Write a function that meets the specification. So it's called char_counts. I've got an input that is a string s, lowercase characters. Assume it's just got vowels and consonants. Return for me a tuple where the first element in the tuple is how many vowels are in s.

And the second element of the tuple is how many consonants are in s. So it should be pretty straightforward. A hint I have here, if you don't remember, that will make your life easier is try to remember how to check if a character is in a string, so using the special n keyword. We saw an example of this probably back when we learned about strings.

So you can try to write your code around line 65-ish. And then we can write it together. All right, so how would you approach this problem? So what's the first step here? Yeah.

AUDIENCE: Make a string that contains all the vowels?

ANA BELL: Yep, we can make a string that contains all the vowels. Vowels equals aeiou in lowercase. Yep, nice. Next? Yeah.

AUDIENCE: If the [INAUDIBLE] in that list, then we could have a number [INAUDIBLE] track, like plus equals [INAUDIBLE]?

ANA BELL: Yep. Yep, vowels plus equals 1. And else? We know it's not a vowel, so we'll keep a consonants count plus equals 1. So this is the consonant count. And this is the vowels count. What is char in this case though?

AUDIENCE: [INAUDIBLE] char In that?

ANA BELL: Yeah, exactly. We have to loop, so for char in s. So we need to look at every character inside s. And this is where, now that we're dealing with things that might be non integers in my for loops, we can write little notes for ourselves that's something like "char is a then b then c," or something like that to remind us that char is not the index, but it's an actual thing. And then what else we need to do?

AUDIENCE: [INAUDIBLE]

ANA BELL: Yeah, we can initiate c and v 0. We can use the trick where you do c comma v equals 0 comma 0. Or we can just do it on separate lines, all good. And then lastly, this does the work for us. But the function needs to have something to show for it.

AUDIENCE: [INAUDIBLE]?

ANA BELL: Yeah, after the loop, we'll return the tuple c comma v-- sorry, v comma c probably. And if we run it, it matches what we expected. So 1, 3 and 0, 5. And you can imagine adding a couple more test cases, maybe something with an empty string that should return 0 0, and maybe something with all vowels, which should return some number comma 0.

OK, so one other thing we can do with tuples is to create these functions that take a variable number of arguments is in as a parameter. So remember, when we define functions, we basically tell Python how many parameter we expect it to take. But it's possible to have some functions, for example, max or min, that can take in two parameters here. And notice, there's no extra parentheses.

Or we can just add as many numbers as we'd like and it will still work to take the max of all of these sets of numbers. And again, we didn't make this inner thing a tuple, although it works even with the tuple as an object. But our goal here is to try to write a function that can take in a variable number of arguments, either two or three or 10 or 20, and it should still work.

And the way we do that is using a parameter that's defined using the star notation. So as soon as you create a function and its parameter is star and then the name of your input, Python basically takes that input and assigns it to a tuple behind the scenes. So you don't have to.

And so, in this particular case, we're not writing our own max or min or some, we're writing our own mean function. And this mean function will take in a variable number of arguments. And it's going to figure out the mean of all of these values. The way it does that is pretty simple now that we know that we can just treat args as a tuple of a bunch of numbers.

So we just loop through all of the elements in args, we add up this running total, and at the end, we return the total divided by how many arguments were given, so return total divided by the length of the args. And then, when we make a Function call to the function we just wrote mean here, args will become a tuple that's all of the parameters inside there.

And so here is that example, which means that we can use our function to get the mean of 1, 2, 3, 4, 5, 6. But we can use the exact same function to get the mean of 6, 0, 9 for example. So first case, I have six parameters as my input. But in the second case, I've got only three parameters as my input. And that little star in my arguments allows me to do this.

Now, I did write a version of this mean for you guys down here where I'm assuming that mean doesn't have the star, so assuming that args is a tuple itself. And in that case, you would have to call the mean function by explicitly passing in only one argument that is a tuple. So this extra set of parentheses makes my argument the tuple. So take a closer look at that if you're interested.

So I want to introduce lists today. And a lot of the slides here are basically copy and paste from the tuple slides. The only difference in these slides that I have regarding lists is the way we define a list.

So in terms of defining a tuple, we were using parentheses. But to define a list, we use open closed square brackets. But otherwise, a lot of the operations are exactly the same as tuples and as strings.

We're not going to look at what it means for lists to be mutable this lecture. But next lecture will be all about mutability. But today, I just want to give you a sense of what a list is. So as I said, this is copy and paste from the tuple slide.

When I create a list, I just use open and close square brackets. This creates a list for me with no elements within it. Creating a list with one element in it doesn't need that extra comma because there's no confusion with operation precedence with square brackets.

So there's no need for that. But otherwise, everything else here is exactly the same as with tuples. We're just using square brackets instead of parentheses. So remember, strings and tuples, it's the same.

What I do want to mention and talk a little bit about, now that we've introduced tuples and lists, is the idea of having our loops iterate over elements of tuples and lists directly. And I'm going to basically write these slides in the context of lists. But the exact same thing is applicable to tuples, as well.

So here is an example of us wanting to find the sum of the elements in a list. The code on the left is a little bit hard to parse, right? We've got a loop variable going through range length n. And then I have to keep my running total.

But I have to index into the list at that index here. And it's really hard to tell what's going on at a quick glance. And so luckily for us, the way that we were able to iterate over string characters directly, we can iterate over tuple and list elements directly.

So the right-hand side here is code that does exactly the same thing as the one on the left, except that our loop variable i, in this particular case, will take on the values of my list directly. And so if we take that code-- yes, and I guess we call this version more "Pythonic" because it's a lot easier to read. So if we take that code and wrap it around the function to make this piece of code be something that we can reuse in a whole bunch of places to grab the sum of all the elements of a list, we can do that.

So here, I've taken the code that does the work, I've plopped it inside this function, I've named list sum, I've taken a list as a parameter, and instead of printing the total, I'm returning the total. So very useful function now. This loop variable i will take on the values 8, then 3, then 5, if that's the list I called this function with, so a lot nicer than iterating over the index and then indexing into the list with the square brackets at that index.

What I do want to mention is something, when you're writing code-- and this is something that I used to do when I first started out-- is to write a little comment for yourself right underneath for loop. Now, I know it's a little tedious, but it does help you keep track of-- especially now that we're iterating over tuples or over lists or over string elements directly, or even over the indices, it helps you keep track of what this loop variable's value is going to be.

And then you don't have to keep track of it in your mind. It's on paper. And you can use your mind to keep track of other things. So if you just write a little comment for yourself there, it helps you debug along the way.

So once we iterate over list elements directly, it makes code that we write really easy to read. So here, the code on the left is going to iterate over the elements directly and get the running total. But we can make a really small change to the input list. Let's say our input list no longer takes in just numbers, but it can take in strings.

We can make one small change to our loop body. Our loop variable still iterates over all the elements in the list l. And then if we write the note for ourselves that s is going to be a, b, then def, then g, if we wanted to write code that grabbed the sum of all of the lengths of the list, the total plus equals e on the left-hand side becomes total plus equal length of s on the right-hand side.

So length of s, one at a time, will take on the value 2 because a, b has length 2, and then 3 because def has length 3, and then 1 because g has length 1. So the code looks very similar, but they have different functionalities depending on what we want to do.

OK, we don't have time to go through this to try it. But definitely try it on your own at home. It's very useful. Plus a whole bunch of other functions that I've put in this Python file for you to get a lot of experience with tuples and lists, OK.