...
Now that we've introduced functions into our language,
we have a lot of power for doing computation.
But so far, all we can do is do computation
on strings and on numbers.
Still a lot of things we can do, but we'd
like to not only have the ability
to do strings and numbers, we'd like
to be able to look at bigger collections of data.
And the same way that functions let us group things together,
common computations, ways of grouping data together
into things we can treat as primitives, as simple things,
is going to be equally powerful.
That's what we're going to talk about today.
We're going to produce two new kinds of data structures,
things called "tuples," and things called "lists."
Let's start with the tuple.
A "tuple" is an ordered sequence of elements
which can include any different kind of element within them.
Now, when I say an ordered sequence,
I don't mean that the elements in the sequence
are ordered, meaning smallest to largest.
I mean that the sequence itself has an order so that I
can get to different parts of the sequence
by simply indexing, much like we did with strings.
One of the issues about tuples is that they're immutable,
meaning I cannot change values inside of them.
And in fact, that's just like strings.
Remember, we could take out parts of a string,
but we couldn't change we couldn't change
the inner pieces of a string.
How do we build tuples?
Well, I've got examples here in this slide.
I'm going to show you some of these inside of a Python
environment.
I can create an empty tuple by simply giving it
an open and close paren.
The parens designate that this is a tuple.
We'll see why that's important in a second.
So te is an empty tuple.
I could make another tuple by giving it
a sequence of elements, the integer 2, the string 1,
the integer 3, for example.
I could print it out.
It's going to give me back the same thing.
And like with strings, I can index into them.
So I can say, starting again at zero,
what is the zeroth element of the tuple?
Because I'm indexing, I'm using square brackets or braces
to get to it.
And that gives me the first element.
As with strings, I can concatenate,
using basically the same kind of structure.
I could take t and add to it another tuple, say 5 and 6.

And that gives me back the longer tuple,

all concatenated together.
And just like with strings, I can slice
to get out portions of a tuple.
However, I can't, as with strings, try and change them.
Because they're immutable.
If I try and change the first element of t,
say I want it to be 4.
It's going to complain.
It gives me a type error, saying that this is not
something that supports that kind of an operation.
There are tuples.
Really good.
OK.
With those, I can now start adding up
other kinds of pieces.
Before I do it though, let me point out
one last thing, which is the funny thing about the example I
just highlighted here.
If I go to t, there it is, and I say, give
me, basically, the first element, so t from 1
but up to 2.
So give me the tuple, I should say,
rather than the first element.
Give me the tuple that basically goes from 1 but not including
the second element.
It gives me back a slightly funky looking thing.
It gives me back an open paren saying, this is a tuple.
It gives me the element.
But then, it gives me a comma before the close paren.
That extra comma is telling me that this is, in fact, a tuple.
To see that, let's look at the difference.
I could create a tuple by saying 1 and a comma.
And it gives me back a tuple.
What would happen if I did open paren and the string
1, close paren?
Oh.
It just treats the parens in this case
as if they're scoping things.
And it just gives me back the element.
So for tuples, we have to give that extra comma
to tell us that, in fact, this is a tuple, and not just
an expression that's going to reduce to a single element.
So there are tuples.
One of the nice things about tuples,
they're really convenient for things like swapping variables.
Again, imagine I've got x and y.
And I want to flip the values.
We saw this before.
The first version won't work because I'm
going to take x and give it the value of y.
And then, when I want to bind y, the value of x is lost.
I could do it by creating a temporary variable.
I've done that before.
Hold the value of x so that I can move y's value into x
and then take that temporary value and put it back into y.
But tuples do it directly.

This is a perfectly legal command in Python
that says, create the tuple with bindings for x and y
by simply taking the opposite versions in that tuple.
That gives me a nice way, for example, not only
to swap variables, it also allows
me to nicely return more than one value from a function.
So if I wanted to give back, for example, the quotient
and remainder of dividing x by y,
I could compute the quotient.
I could compute the remainder.
Return's only going to return one thing.
But if I return the tuple of q and r,
then I can bind it using an expression that has,
on the left-hand side, a tuple of names.
That will give Quot the value that q holds.
It will give Rem the value that r holds.
So tuples have some really nice properties.
In particular, it's letting us think about a tuple
as a single construct out of which I can pull out
the pieces.
Great.

Another nice thing about tuples, they're iterable.
Just like with strings, where I can walk down them,
I could do the same thing with a tuple.
Here's a little procedure.
I'm going to walk you through it.
It's kind of a funky procedure.
The idea is that I'm going to give GetData a tuple.
And that tuple is going to itself consist of tuples.
Weird.
But it says each element inside that tuple
is itself a collection of things.
In particular, it's going to be a collection of ints
and strings.
And what I want to GetData to do is to create two outputs.
So let's look at it.
First, I'm going to set up Nums to be an empty tuple.
And that's how I do it.
Open close paren.
Words is also going to be an empty tuple.
And I'm going to take all of the numbers
and gather them together into a new tuple.
I'm going to take that in the following way.
Here's my little loop.
Notice, I'm iterating over the tuple.
I'm saying "for t in the tuple."
That says the first time through t
is going to be pointing to that.
Second time through, that.
Third time through, that.
And what do I want to do?
I want to get the number out.
So I'm going to index in the first, or in this case,
I should say it better, the zeroth element of t
gives me that thing right there.

And then, the paren would just treat it
as if it was a single expression.
So the comma, as we've already seen,
is important because it's going to give me that singleton
tuple.
I'm going to add it into Nums.
And add here says, concatenate.
So Nums is now a little longer tuple.
Then, I'm going to for Words say, if the word part,
this part, if it's not inside things
I've already gathered together, I'm going to do the same thing.
I'm basically gathering up the unique words, as well as
all of the integers.
And so they'll get put into there.
And when I'm done, I'm going to return the smallest
number, the largest number, and the number of unique words
that I've found.
That's what the bottom part does.
And if I were to run it, I've got a little example over here,
I simply load that into my environment.
And it didn't print anything out.
But that's OK because my binding is,
I've now bound Small to be the first part of what
was returned.
And that is number 1, which was the smallest number.
Large is the largest number, 7.
And Words should be the unique words,
or the number of unique words, and there were three of them.
So I simply ran that on a set of data.
And what I wanted for you to see here
is how I can iterate over the tuples, treating them
as if they're just a single construct,
just the same I would with a range,
or I would with a string.