...
Let's think about what we've been doing so far.
We started introducing elements to our language.
We started off with simple things-- basic objects, ways
to do arithmetic operations on them, ways to compare them.
We introduced conditionals as a consequence of that.
And we started looking at the first version of ways
to actually put things together into pieces of code.
So we've covered language mechanisms.
We've also covered the first notion of for and while loops
and, therefore, of iteration.
And we've also seen that we could
write different kinds of computations,
each one in a different file that I save away on my machine,
so I can use when I want to do something with it.
Each file contains a piece of code.
And each piece of code is some sequence
of operations-- could be a simple branching program.
It could be an iteration like a for loop or a while loop,
could be something else.
This is fine when our code consists of a few lines.
This is fine when our problems are very small.
But it doesn't really scale very nicely.
As I start thinking about pieces of code that
might be hundreds or thousands of lines long, simply doing it
this way is a real challenge.
It's hard to keep track of the details.
It's hard to make sure that I've got
the right information in the right place in that code.
Does this go at line 95, or at line 296?
I need a way to really capture this better.
I need a way of structuring my code,
structuring my computation, so that I can think about it
much more efficiently.
And that's what we're going to introduce now.
One of the things to keep in mind
is, in fact, good programming is more than just adding
more code.
In fact, I would argue that a good programmer is measured
not by the number of lines of code she's written,
but by the amount of functionality that she's
provided-- the ability to do computations easily.
And so to make that happen, we're
going to take a really nice pivot, introduce
a major new thing, and that's the idea of a function.
We're going to describe it in a second.
But the idea of a function is it's
going to give me a way to encapsulate
pieces of computation.
And in fact, a function is intended
to give me two really important aspects
of computational thinking-- decomposition,
sometimes also called modularity, and abstraction.
So what do those mean?
Let me give you an analogy-- something
a little bit different.

I'm sure you've all seen a projector.
I think of a projector as a black box.
Now, the one I'm showing here happens
to have a black box around it, but my black box,
I literally mean a construct whose details
I don't understand.
In fact, most of us don't know what's inside it,
how it actually works.
If you open it up, you see a bunch of electronics.
And unless you're a really good electrical engineer,
you probably don't understand how it works.
It doesn't matter, because you know the interface,
you know the input-output behavior.
And in fact, that black box comes
with a standard interface, that says,
if I plug an appropriate piece of electronics
into it, that box can communicate with it.
And it's going to produce an output.
It's going to show slides on the wall.
That's the notion of abstraction.
It's the idea that once I've built something-- in this case
a projector, might be a piece of code--
once I've built something, I don't
need to know what's inside it as long as I know how it works.
So abstraction, in some sense, comes
with a contract that says, if you give me appropriate inputs,
I'm going to behave in an appropriate way.
That's a really nice idea.
That's something that we're going to want to use.
And we're going to use it with code.
There's a second piece to it.
And we'll stick with the projector idea.
If you happened to see the opening
ceremonies for the Vancouver, Olympics,
there was a wonderful display in which
there was a huge multi-story-high structure
in the middle of the rink onto which images were projected.
Now, it wasn't one projector.
It was way too big.
It was a sequence of projectors.
Each of them took an input.
Each of them produced an output.
And they all worked together in synchrony
to produce that wonderful larger image.
That's the second big idea, decomposition,
that I can, in this case, take different devices,
have them synchronize, so that they work together
to achieve an end goal.
I want to use the same thing with code.
So we're going to apply those ideas to programming.
Decomposition-- I'm going to break a problem up
into different self-contained pieces.
And once I've built them-- or even better, once somebody else
has built one of them for me, all I need to know
is, what does it expect as input,
and what will the behavior be when it gives me an output?

And the second big idea, abstraction,
that I want to be able to suppress
the details inside that method in order
to be able to compute something using that computation.
And in fact, I just said them in the wrong way.
What I first described was really abstraction.
Decomposition is that idea of breaking it up
into smaller pieces.
With those two things together, we're going to now look at,
how do we make programs have those two properties
Inside of code with decomposition,
we're going to do the same thing we
did when we talked about having multiple projectors working
together.
When I'm going to structure a big piece of code,
I'm going to divide it up into modules.
And those have the property that they're self-contained.
That means, they have everything they need inside of them.
I'm going to use it to break the code up into pieces that I
can separate apart.
And they're intended to be re-usable.
I can use them multiple places.
If you think about it, so far, even if I have a little file,
I can only use it once.
Or if I want to reuse it, I've got
to go back and recall the file.
Here, I'm going to structure code so that I
can use it in multiple places.
And we're going to see that this idea of dividing code
into modules is both going to keep the code organized
and coherent.
It's going to be easier to structure.
Today, we're going to do decomposition with functions.
And in a few weeks time, we're going
to see how to do decomposition with classes,
an equally powerful idea.
We're going to suppress details with abstraction.
Again, I said with the projector,
I don't need to know what's inside.
I just need to know what it does.
In programming, we're also going to think of a piece of code
as a black box.
I don't need to see the details inside once I've written it,
or especially if somebody else has written it for me.
I, in fact, don't want to see the details inside.
What I simply want to know is that it's
going to work the way it was advertised.
And we're going to achieve abstraction
with something called a function specification or a doc string.
And in a second, we're going to see examples of both of those.
But there is the idea-- abstraction, decomposition,
two powerful ideas that we want to capture
inside a computation.
One of the things that comes out of this
is, not only do they work well together,
but that can then be used many times.

And in fact, not only can they be used many times,
I only have to debug it once.
It's one piece of code.
If I want to make a change to it,
I change that piece of code.
Anything else that depends on that code
will inherit that behavior, because they simply
depend on that code.