PROFESSOR: Hi. Today I'd like to talk to you about state machines. State machines are an incredibly important concept, both in 6.01, and in the general sense when you're going to be doing things like control theory or artificial intelligence. Any further work you do in terms of computability theory, state machines are really important.

So I'm going to review what we've done so far and talk about why we now need state machines to add additional complexity to the kinds of models that we're going to build, and then talk a little bit about how state machines are represented in different domains, and then also talk about how we're going to use state machines in the 6.01 software.

First of all, let's review what we've learned so far. We've talked, so far, about functional, imperative and object-oriented programming paradigms. In functional programming, everything is a function. And in imperative programming, we're allowed to use functions, but they're also allowed to have side effects. And in object-oriented programming, everything is an object.

We can actually use the first two to implement the last one, and use the first one plus an idea of maintaining assignments to variables, that sort of thing, to implement this one. So you can trace the progression of different kinds of computer science language along this paradigm.

But the one thing that none of these languages on their own allows us to do is keep a notion of internal state. And what I mean by that is that if we have a system that we want to model in terms of the passage of time or keep track of the evolution of that system or keep track of some of the data that has accumulated over time in that system, then we certainly can't do it with functional programming. Functional programming takes one input, generates one output, and you could generate a list of codes that took in every possible situation and then generated the logical output, but that would be a lot of code.

Same thing for imperative and object-oriented programming. They alone do not address everything that we want to address, which is the ability to look at everything that has happened as a consequence of the passage of time and all of the data that we've looked at as a consequence of the passage of time, synthesize it in some way, and then generate whatever was supposed to come out of that situation. That is the notion of internal state.

State machines have been around for a while. You might see them refer to as discrete finite automata. There are also such things as continuous state machines, but we're not going to talk about those in this course so much. So we're only talking about discrete state machines.

And when you see them referred to as state machines or discrete finite automata in literature, especially mathematics, you'll be looking for this five set of things. One is a set of the possible states you could be in, given a particular state machine. One is the set of inputs you could possibly encounter while in that state machine. One is the set of outputs that that state machine could possibly generate. One is a transition function that looks at pairs of these values and tells you, based on which state you're currently in, what state you will end up in as a consequence of the current input, and then also specify what the output will be as a consequence of that transition. And finally, it'll tell you where you start out.

That's a lot to absorb. I'm going to show you a state transition diagram, which many of you have probably seen before, and map these values to this state transition diagram in hopes of making this a little bit more concrete.

Hopefully at this point, all of you who have interacted with an MBTA turnstile. This is a thing that opens and closes, and you stick your RFID card on it, and then Richard Stallman yells at you and possibly hands you more Charlie tickets or something like that.

It has four states. One of them is that this turnstile could be closed, and it's waiting for people to interact with it in some way. It could be open, and I anthropomorphize the turnstile as being happy as a consequence of you putting money in it. It could be open and quiet as a consequence of usually some sort of other previous interaction with another person. Or it could be open and angry as a consequence of people interacting with it when they should not.

The vertices in this directed graph are my set of states. So when you see something like this, and you're asked to map it to the mathematical construct, just grab the names and say this is my set of states.

Let's say I start off in closed. And actually, it occurs to me that the other thing that should be specified is a start state. And it's not here. So let's say the turnstile starts off as closed. Usually this is an arrow coming out of nowhere that directs into one of the states. Sometimes it will be explicitly indicated by saying start state, but typically you'll just see an arrow from nowhere.

At this point, I don't have any inputs or outputs. If I put money in the turnstile, it constitutes input to my system. It's going to end up in my set of inputs for math. My transition function looks at the state I'm in and looks at the current input and generates the output and the next state. So all of these arrows, in addition to whatever information is contained in the receiving vertex, specifies my transition function.

Any transition that's not specified is not considered in the function. This was not part of our original drawing. So if I was fed open angry, there would be no way to get to open happy. Likewise, if I was in open angry and fed money for this simple system, we're going to say nothing would happen.

I think at this point it's become clear how to transform a state transition diagram into the mathematical constructs. It's good to have the mathematical constructs, because they end up being used in software, which I'll talk about in a second. But the first thing I'm going to do is just walk around the state transition diagram and indicate what would be inputs, what would be outputs, that sort of thing.

So let's say I walk up to the turnstile and somebody else interacts with the turnstile by exiting. In this case, exit is the input, none is the output, the turnstile doesn't make any noise, and the turnstile is open. If at that point I interact with the turnstile by entering, it's going to make noise, which is the output. And then the new state is going to be that the turnstile is open and angry.

At that point, you and I know that the turnstile is going to close. So this edge indicates that the only available input at that point is to do nothing. Or independent of anything else, it's going to squawk again and close.

One more time, here are the states. Inputs are the first of these two pairs. Outputs, as a consequence of the transition, is the second of these two pairs. And the transition function is represented by the directed edge and the new state.

Once you have all those sets figured out, you can start talking about how to implement state machines and software. We've actually abstracted this away from you. You don't have to deal with it. But as a consequence, you should know how to interact with the 6.01 library.

Let's look at an example of the state machine class. I want to build an accumulator, which at every time step we'll look at the input, add it to every other example of input and output, and output the new value and retain it as the new state. The first thing I need to do is initialize the accumulator with a value. This is our start state, which is the same as our start state from the MBTA turnstile, and also the same as our start state from the mathematical construct.

I also want something called getNext Values which is the functional equivalent of the transitions. Here's our self again from object oriented programming -- but we don't care about that. We're going to look at the current state and the current input. Some getNext Values we'll do some internal data munging, possibly multiplication by two or comparing to the previous input and then having some conditionals, that sort of thing. But there'll be some sort of very simple function under getNext Values at least for the first couple of weeks.

And then we will return the new state and the output into tuple. In this case, the new state is going to be the linear combination of the current state and the current input. And the output is going to be the linear combination of the current state and the current input.

If I were to draw this accumulator as a state transition diagram, I would do this. My start state is the initial value. If I pass in a new input-- we'll call it input 0 -- both my output and the new state are going to be the linear combination of these two values. If I made another transition, I would take whatever my next input was and add it to the current state value and return it out as the output, and so on and so forth.

I encourage you try this in Python, or in IDLE and munge around with it and see what you can get it to do. You might have to type add lib6.01 in order to get the state machine class, or initialize using lib6.01 in order to get the state machine class. But otherwise, those should be enough to get you started with an introduction to state machines.

If you're having trouble, I highly recommend going through all the examples in the readings. They're pretty comprehensive, and it also includes the accumulator. That's all for now. Next time we'll talk about linear time varying systems.