ANA BELL: All right. Let's begin. As I mentioned before, this lecture will be recorded for OCW. Again, in future lectures, if you don't want to have the back of your head show up, just don't sit in this front area here.

First of all, wow, what a crowd, you guys. We're finally in 26-100. 6.0001 made it big, huh? Good afternoon and welcome to the very first class of 6.0001, and also 600, this semester.

My name is Ana Bell. First name, Ana. Last name, Bell. I'm a lecturer in the EECS Department. And I'll be giving some of the lectures for today, along with later on in the term, Professor Eric Grimson, who's sitting right down there, will be giving some of the lectures, as well.

Today we're going to go over some basic administrivia, a little bit of course information. And then, we're going to talk a little bit about what is computation? We'll discuss at a very high level what computers do just to make sure we're all on the same page.

And then, we're going to dive right into Python basics. We're going to talk a little bit about mathematical operations you can do with Python. And then, we're going to talk about Python variables and types.

As I mentioned in my introductory email, all the slides and code that I'll talk about during lectures will be up before lecture, so I highly encourage you to download them and to have them open. We're going to go through some in-class exercises which will be available on those slides. And it's fun to do.

And it's also great if could take notes about the code just for future reference. It's true. This is a really fast-paced course, and we ramp up really quickly. We do want to position you to succeed in this course.

As I was writing this, I was trying to think about when I was first starting to program what helped me get through my very first programming course. And this is really a good list. The first thing was I just read the psets as soon as they came out, made sure that the terminology

just sunk in.

And then, during lectures, if the lecturer was talking about something that suddenly I remembered, oh, I saw that word in the pset and I didn't know what it was. Well, hey, now I know what it is. Right? So just give it a read. You don't need to start it.

If you're new to programming, I think the key word is practice. It's like math or reading. The more you practice, the better you get at it. You're not going to absorb programming by watching me write programs because I already know how to program. You guys need to practice.

Download the code before lecture. Follow along. Whatever I type, you guys can type. And I think, also, one of the big things is if you're new to programming, you're kind of afraid that you're going to break your computer. And you can't really do that just by running Anaconda and typing in some commands.

So don't be afraid to just type some stuff in and see what it does. Worst case, you just restart the computer. Yeah. That's probably the big thing right there. I should have probably highlighted it, but don't be afraid.

Great. So this is pretty much a roadmap of all of 6.0001 or 600 as I've just explained it. There's three big things we want to get out of this course. The first thing is the knowledge of concepts, which is pretty much true of any class that you'll take.

The class will teach you something through lectures. Exams will test how much you know. This is a class in programming. The other thing we want you to get out of it is programming skills.

And the last thing, and I think this is what makes this class really great, is we teach you how to solve problems. And we do that through the psets. That's really how I feel the roadmap of this course looks like.

And underlying all of these is just practice. You have to just type some stuff away and code a lot. And you'll succeed in this course, I think.

OK. So what are the things we're going to learn in this class? I feel like the things we're going learn in this class can be divided into basically three different sections. The first one is related to these first two items here. It's really about learning how to program.

Learning how to program, part of it is figuring out what objects to create. You'll learn about these later. How do you represent knowledge with data structures? That's sort of the broad term for that.

And then, as you're writing programs, you need to-- programs aren't just linear. Sometimes programs jump around. They make decisions. There's some control flow to programs. That's what the second line is going to be about.

The second big part of this course is a little bit more abstract, and it deals with how do you write good code, good style, code that's readable. When you write code, you want to write it such that-- you're in big company, other people will read it, other people will use it, so it has to be readable and understandable by others.

To that end, you need to write code that's well organized, modular, easy to understand. And not only that, not only will your code be read by other people, but next year, maybe, you'll take another course, and you'll want to look back at some of the problems that you wrote in this class.

You want to be able to reread your code. If it's a big mess, you might not be able to understand-- or reunderstand-- what you were doing. So writing readable code and organizing code is also a big part.

And the last section is going to deal with-- the first two are actually part of the programming in Introduction to Programming and Computer Science in Python. And the last one deals mostly with the computer science part in Introduction to Programming and Computer Science in Python.

We're going to talk about, once you have learned how to write programs in Python, how do you compare programs in Python? How do you know that one program is better than the other?

How do you know that one program is more efficient than the other? How do you know that one algorithm is better than the other? That's what we're going to talk about in the last part of the course.

OK. That's all for the administrative part of the course. Let's start by talking at a high level what does a computer do.

Fundamentally, it does two things. One, performs calculations. It performs a lot of calculations. Computers these days are really, really fast, a billion calculations per second is probably not far off. It performs these calculations and it has to store them somewhere. Right? Stores them in computer memory.

So a computer also has to remember results. And these days, it's not uncommon to find computers with hundreds of gigabytes of storage. The kinds of calculations that computers do, there are two kinds.

One are calculations that are built into the language. These are the very low level types of calculations, things like addition, subtraction, multiplication, and so on.

And once you have a language that has these primitive calculation types, you, as a programmer, can put these types together and then define your own calculations. You can create new types of calculations. And the computer will be able to perform those, as well.

I think, one thing I want to stress-- and we're going to come back to this again during this entire lecture, actually-- is computers only know what you tell them. Computers only do what you tell them to do. They're not magical. They don't have a mind.

They just know how to perform calculations really, really quickly. But you have to tell them what calculations to do. Computers don't know anything. All right. We've come to that.

Let's go into the types of knowledge. The first type of knowledge is declarative knowledge. And those are things like statements of fact. And this is where my email came into play. If you read it all the way to the bottom, you would have entered a raffle.

So a statement of fact for today's lecture is, someone will win a prize before class ends. And the prize was a Google Cardboard. Google state-of-the-art virtual reality glasses. And I have them right here. Yea. I delivered on my promise.

That's a statement of fact. So pretend I'm a machine. OK? I don't know anything except what you tell me. I don't know. I know that you tell me this statement. I'm like, OK. But how is someone going to win a Google Cardboard before class ends, right?

That's where imperative knowledge comes in. Imperative knowledge is the recipe, or the how-to, or the sequence of steps. Sorry. That's just my funny for that one. So the sequence of steps is imperative knowledge.

If I'm a machine, you need to tell me how someone will win a Google Cardboard before class. If I follow these steps, then technically, I should reach a conclusion.

Step one, I think we've already done that. Whoever wanted to sign up has signed up. Now I'm going to open my IDE. I'm just basically being a machine and following the steps that you've told me.

The IDE that we're using in this class is called Anaconda. I'm just scrolling down to the bottom. Hopefully, you've installed it in problem set zero. I've opened my IDE. I'm going to follow the next set of instructions. I'm going to choose a random number between the first and the nth responder.

Now, I'm going to actually use Python to do this . And this is also an example of how just a really simple task in your life, you can use computers or programming to do that. Because if I chose a random number, I might be biased because, for example, I might like the number 8.

To choose a random number, I'm going to go and say, OK, where's the list of responders? It starts at 15. Actually, it starts at 16 because that's me. We're going to choose a random number between 16 and the end person 266. Oh, we just got-- oh. OK.

OK. I'm going to cut it off right here. 271. OK. 16 and 271. Perfect. OK. I'm going to choose a random number. I'm going to go to my IDE. And you don't need to know how to do this yet, but by the end of this class, you will. I'm just going to use Python.

I'm just going to get the random number package that's going to give me a random number. I'm going to say random.randint. And I'm going to choose a random number between 16 and 272,

OK. 75. OK. Great. I chose a random number. And I'm going to find the number in the responder's sheet. What was the number again? Sorry. 75. OK. Up we go. There we go. Lauren Z-O-V. Yeah. Nice. You're here.

Awesome. All right. That's an example of me being a machine and also, at the same time, using Python in my everyday life, just lecturing, to find a random number. Try to use Python wherever you can. And that just gives you practice.

That was fun. But we're at MIT. We're MIT students. And we love numbers here at MIT. Here's a numerical example that shows the difference between declarative and imperative

knowledge.

An example of declarative knowledge is the square root of a number x is y such that y times y is equal to x. That's just a statement of fact It's true. Computers don't know what to do with that. They don't know what to do with that statement. But computers do know how to follow a recipe.

Here's a well-known algorithm. To find the square root of a number x, let's say x is originally 16, if a computer follows this algorithm, it's going to start with a guess, g, let's say, 3. We're trying to find the square root of 16.

We're going to calculate g times g is 9. And we're going to ask is if g times g is close enough to x, then stop and say, g is the answer.

I'm not really happy with 9 being really close to 16. So I'm going to say, I'm not stopping here. I'm going to keep going.

If it's not close enough, then I'm going to make a new guess by averaging g and x over g. That's x over g here. And that's the average over there.

And the new average is going to be my new guess. And that's what it says. And then, the last step is using the new guess, repeat the process. Then we go back to the beginning and repeat the whole process over and over again.

And that's what the rest of the rows do. And you keep doing this until you decide that you're close enough. What we saw for the imperative knowledge in the previous numerical example was the recipe for how to find the square root of x. What were the three parts of the recipe?

One was a simple sequence of steps. There were four steps. The other was a flow of control, so there were parts where we made decisions. Are we close enough? There were parts where we repeated some steps. At the end, we said, repeat steps 1, 2, 3. That's the flow of control.

And the last part of the recipe was a way to stop. You don't want a program that keeps going and going. Or for a recipe, you don't want to keep baking bread forever. You want to stop at some point. Like 10 breads is enough, right? So you have to have a way of stopping.

In the previous example, the way of stopping was that we decided we were close enough. Close enough was maybe being within .01, .001, whatever you pick. This recipe is there for an

algorithm. In computer science speak, it's going to be an algorithm. And that's what we're going to learn about in this class.

We're dealing with computers. And we actually want to capture a recipe inside a computer, a computer being a mechanical process. Historically, there were two different types of computers. Originally, there were these things called fixed-program computers.

And I'm old enough to have used something like this, where there's just numbers and plus, minus, multiplication, divide, and equal. But calculators these days are a lot more complicated.

But way back then, an example of a fixed-program computer is this calculator. It only knows how to do addition, multiplication, subtraction, division. If you want to plot something, you can't. If you want to go on the internet, send email with it, you can't.

It can only do this one thing. And if you wanted to create a machine that did another thing, then you'd have to create another fixed-program computer that did a completely separate test. That's not very great.

That's when stored-program computers came into play. And these were machines that could store a sequence of instructions. And these machines could execute the sequence of instructions. And you could change the sequence of instructions and execute this different sequence of instructions.

You could do different tasks in the same machine. And that's the computer as we know it these days. The central processing unit is where all of these decisions get made. And these are all the peripherals.

The basic machine architecture-- at the heart of every computer there's just this basic architecture-- and it contains, I guess, four main parts. The first is the memory. Input and output is the other one.

The ALU is where all of the operations are done. And the operations that the ALU can do are really primitive operations, addition, subtraction, and so on.

What the memory contains is a bunch of data and your sequence of instructions. Interacting with the Arithmetic Logic Unit is the Control Unit. And the Control Unit contains one program counter.

When you load a sequence of instructions, the program counter starts at the first sequence. It starts at the sequence, at the first instruction. It gets what the instruction is, and it sends it to the ALU.

The ALU asks, what are we doing operations on here? What's happening? It might get some data. If you're adding two numbers, it might get two numbers from memory. It might do some operations. And it might store data back into memory.

And after it's done, the ALU is going to go back, and the program counter is going to increase by 1, which means that we're going to go to the next sequence in the instruction set. And it just goes linearly, instruction by instruction.

There might be one particular instruction that does some sort of test. It's going to say, is this particular value greater or equal to or the same as this other particular value? That's a test, an example of a test. And the test is going to either return true or false.

And depending on the result of that test, you might either go to the next instruction, or you might set the program counter to go all the way back to the beginning, and so on. You're not just linearly stepping through all the instructions. There might be some control flow involved, where you might skip an instruction, or start from the beginning, or so on.

And after you're done, when you finished executing the last instruction, then you might output something. That's really the basic way that a computer works. Just to recap, you have the stored program computer that contains these sequences of instructions.

The primitive operations that it can do are addition, subtraction, logic operations, tests-- which are something equal to something else, something less than, and so on-- and moving data, so storing data, moving data around, and things like that.

And the interpreter goes through every instruction and decides whether you're going to go to the next instruction, skip instructions, or repeat instructions, and so on.

So we've talked about primitives. And in fact, Alan Turing, who was a really great computer scientist, he showed that you can compute anything using the six primitives. And the six primitives are move left, move right, read, write, scan, and do nothing.

Using those six instructions and the piece of tape, he showed that you can compute anything. And using those six instructions, programming languages came about that created a more

convenient set of primitives. You don't have to program in only these six commands.

And one interesting thing, or one really important thing, that came about from these six primitives is that if you can compute something in Python, let's say-- if you write a program that computes something in Python, then, in theory, you can write a program that computes the exact same thing in any other language. And that's a really powerful statement.

Think about that today when you review your slides. Think about that again. That's really powerful. Once you have your set of primitives for a particular language, you can start creating expressions. And these expressions are going to be combinations of the primitives in the programming language.

And the expressions are going to have some value. And they're going up some meaning in the programming language. Let's do a little bit of a parallel with English just so you see what I mean. In English, the primitive constructs are going to be words. There's a lot of words in the English language.

Programming languages-- in Python, there are primitives, but there aren't as many of them. There are floats, Booleans, these are numbers, strings, and simple operators, like addition, subtraction, and so on. So we have primitive constructs.

Using these primitive constructs, we can start creating, in English, phrases, sentences, and the same in programming languages. In English, we can say something like, "cat, dog, boy. That, we say, is not syntactically valid. That's bad syntax. That's noun, noun, noun. That doesn't make sense.

What does have good syntax in English is noun, verb, noun. So, "cat, hugs boy" is syntactically valid. Similarly, in a programming language, something like this-- in Python, in this case-- a word and then the number five doesn't really make sense. It's not syntactically valid. But something like operator, operand, operator is OK.

So once you've created these phrases, or these expressions, that are syntactically valid, you have to think about the static semantics of your phrase, or of your expression. For example, in English, "I are hungry" is good syntax.

But it's weird to say. We have a pronoun, a verb, and an adjective, which doesn't really make sense. "I am hungry" is better. This does not have good static semantics.

Similarly, in programming languages-- and you'll get the hang of this the more you do it-- something like this, "3.2 times 5, is OK. But what does it mean? What's the meaning to have a word added to a number? There's no meaning behind that.

Its syntax is OK, because you have operator, operand, operator. But it doesn't really make sense to add a number to a word, for example.

Once you have created these expressions that are syntactically correct and static, semantically correct, in English, for example, you think about the semantics. What's the meaning of the phrase? In English, you can actually have more than one meaning to an entire phrase.

In this case, "flying planes can be dangerous" can have two meanings. It's the act of flying a plane is dangerous, or the plane that is in the air is dangerous.

And this might be a cuter example. "This reading lamp hasn't uttered a word since I bought it. What's going on?" So that has two meanings. It's playing on the word "reading lamp."

That's in English. In English, you can have a sentence that has more than one meaning, that's syntactically correct and static, semantically correct. But in programming languages, the program that you write, the set of instructions that you write, only has one meaning. Remember, we're coming back to the fact that the computer only does what you tell it to do.

It's not going to suddenly decide to add another variable for some reason. It's just going to execute whatever statements you've put up. In programming languages, there's only one meaning.

But the problem that comes into play in programming languages is it's not the meaning that you might have intended, as the programmer. That's where things can go wrong.

And there's going to be a lecture on debugging a little bit later in the course. But this is here just to tell you that if you see an error pop up in your program, it's just some text that says, error. For example, if we do something like this, this is syntactically correct. Incorrect. Syntactically incorrect. See? There's some angry text right here. What is going on?

The more you program, the more you'll get the hang of reading these errors. But this is basically telling me the line that I wrote is syntactically incorrect. And it's pointing to the exact line and says, this is wrong, so I can go back and fix it as a programmer.

Syntax errors are actually really easily caught by Python. That was an example of a syntax error. Static semantic errors can also be caught by Python as long as, if your program has some decisions to make, as long as you've gone down the branch where the static semantic error happens.

And this is probably going to be the most frustrating one, especially as you're starting out. The program might do something different than what you expected it to do. And that's not because the program suddenly-- for example, you expected the program to give you an output of 0 for a certain test case, and the output that you got was 10.

Well, the program didn't suddenly decide to change its answer to 10. It just executed the program that you wrote. That's the case where the program gave you a different answer than expected.

Programs might crash, which means they stop running. That's OK. Just go back to your code and figure out what was wrong. And another example of a different meaning than what you intended was maybe the program won't stop. It's also OK. There are ways to stop it besides restarting the computer.

So then Python programs are going to be sequences of definitions and commands. We're going to have expressions that are going to be evaluated and commands that tell the interpreter to do something.

If you've done problem set 0, you'll see that you can type commands directly in the shell here, which is the part on the right where I did some really simple things, 2 plus 4. Or you can type commands up in here, on the left-hand side, and then run your program.

Notice that, well, we'll talk about this-- I won't talk about this now. But these are-- on the right-hand side, typically, you write very simple commands just if you're testing something out. And on the left-hand side here in the editor, you write more lines and more complicated programs.

Now we're going to start talking about Python. And in Python, we're going to come back to this, everything is an object. And Python programs manipulate these data objects. All objects in Python are going to have a type.

And the type is going to tell Python the kinds of operations that you can do on these objects. If an object is the number five, for example, you can add the number to another number,

subtract the number, take it to the power of something, and so on.

As a more general example, for example, I am a human. So that's my type. And I can walk, speak English, et cetera. Chewbacca is going to be a type Wookie. He can walk, do that sound that I can't do. He can do that, but I can't. I'm not even going to try, and so on.

Once you have these Python objects, everything is an object in Python. There are actually two types of objects. One are scalar objects. That means these are very basic objects in Python from which everything can be made. These are scalar objects. That can't be subdivided.

The other type of object is a non-scalar object. And these are objects that have some internal structure. For example, the number five is a scalar object because it can't be subdivided.

But a list of numbers, for example, 5, 6, 7,8, is going to be a non-scalar object because you can subdivide it. You can subdivide it into-- you can find parts to it. It's made up of a sequence of numbers.

Here's the list of all of the scalar objects in Python. We have integers, for example, all of the whole numbers. Floats, which are all of the real numbers, anything with a decimal.

Bools are Booleans. There's only two values to Booleans. That's True and False. Note the capitalization, capital T and capital F. And this other thing called NoneType. It's special. It has only one value called None. And it represents the absence of a type. And it sometimes comes in handy for some programs.

If you want to find the type of an object, you can use this special command called type. And then in the parentheses, you put down what you want to find the type of. You can write into the shell "type of 5," and the shell will tell you, that's an integer.

If you happen to want to convert between two different types, Python allows you to do that. And to do that, you put the type that you want to convert to right before the object that you want to convert to. So float(3) will convert the integer 3 to the float 3.0.

And similarly, you can convert any float into an integer. And converting to an integer just truncates. It just takes away the decimal and whatever's after it-- it does not round-- and keeps just the integer part.

For this slide, I'm going to talk about it. But if you'd like if you have the slides up, go to go to

this exercise. And after I'm done talking about the slide, we'll see what people think for that exercise.

One of the most important things that you can do in basically any programming, in Python also, is to print things out. Printing out is how you interact with the user.

To print things out, you use the print command. If you're in the shell, if you simply type "3 plus 2," you do see a value here. Five, right? But that's not actually printing something out.

And that becomes apparent when you actually type things into the editor. If you just do "3 plus 2," and you run the program-- that's the green button here-- you see on the right-hand side here, it ran my program. But it didn't actually print anything.

If you type this into the console, it does show you this value, but that's just like peeking into the value for you as a programmer. It's not actually printing it out to anyone. If you want to print something out, you have to use the print statement like that. In this case, this is actually going to print this number five to the console.

That's basically what it says. It just tells you it's an interaction within the shell only. It's not interacting with anyone else. And if you don't have any "Out," that means it got printed out to the console.

All right. We talked a little bit about objects. Once you have objects, you can combine objects and operators to form these expressions. And each expression is going to have a value. So an expression evaluates to a value. The syntax for an expression is going to be object, operator, object, like that.

And these are some operators you can do on ints and floats. There's the typical ones, addition, subtraction, multiplication, and division. If, for the first three, the answer that you get-- the type of the answer that you get-- is going to depend on the type of your variables. If both of the variables of the operands are integers, then the result you're going to get is of type integer.

But if at least one of them is a float, then the result you're going to get is a float. Division is a little bit special in that no matter what the operands are, the result is always going to be a float.

The other operations you can do, and these are also useful, are the remainder, so the percent sign. If you use the percent sign between two operands, that's going to give you the remainder

when you divide i by j.

And raising something to the power of something else is using the star star operator. And i star stars j is going to take i to the power of j.

These operations have the typical precedence that you might expect in math, for example. And if you'd like to put precedence toward some other operations, you can use parentheses to do that.

All right. So we have ways of creating expressions. And we have operations we can do on objects. But what's going to be useful is to be able to save values to some name. And the name is going to be something that you pick.

And it should be a descriptive name. And when you save the value to a name, you're going to be able to access that value later on in your program. And that's very useful.

To save a value to a variable name, you use the equal sign. And the equal sign is an assignment. It assigns the right-hand side, which is a value, to the left-hand side, which is going to be a variable name. In this case, I assigned the float 3.14159 to the variable pi.

And in the second line, I'm going to take this expression, 22 divided by 7, I'm going to evaluate it. It's going to come up with some decimal number. And I'm going to save it into the variable pi_approx. values are stored in memory. And this assignment in Python, we say the assignment binds the name to the value.

When you use that name later on in your program, you're going to be referring to the value in memory. And if you ever want to refer to the value later on in your code, you just simply type the name of the variable that you've assigned it to.

So why do we want to give names to expressions? Well, you want to reuse the names instead of the values. And it makes your code look a lot nicer. This is a piece of code that calculates the area of a circle. And notice, I've assigned a variable pi to 3.14159. I've assigned another variable called radius to be 2.2.

And then, later on in my code, I have another line that says area-- this is another variable-- is equal to-- this is an assignment-- to this expression. And this expression is referring to these variable names, pi and radius.

And it's going look up their values in memory. And it's going to replace these variable names with those values. And it's going to do the calculation for me. And in the end, this whole expression is going to be replaced by one number. And it's going to be the float.

Here's another exercise, while I'm talking about the slide. I do want to make a note about programming versus math.

In math, you're often presented with a problem that says, solve for x. x plus y is equal to something something. Solve for x, for example. That's coming back to the fact that computers don't know what to do with that. Computers need to be told what to do.

In programming, if you want to solve for x, you need to tell the computer exactly how to solve for x. You need to figure out what formula you need to give the computer in order to be able to solve for x.

That means always in programming the right-hand side is going to be an expression. It's something that's going to be evaluated to a value. And the left-hand side is always a variable. It's going to be an assignment.

The equal sign is not like in math where you can have a lot of things to the left and a lot of things to the right of the equal sign. There's only one thing to the left of the equal sign. And that's going to be a variable. An equal sign stands for an assignment.

Once we've created expressions, and we have these assignments, you can rebind variable names using new assignment statements. Let's look at an example for that. Let's say this is our memory. Let's type back in the example with finding the radius.

Let's say, pi is equal to 3.14. In memory, we're going to create this value 3.14. We're going to bind it to the variable named pi. Next line, radius is equal to 2.2. In memory, we're creating this value 2.2. And we're going to bind it to the variable named radius.

Then we have this expression here. It's going to substitute the values for pi from memory and the value for radius from memory. It's going to calculate the value that this expression evaluates to.

It's going to pop that into the memory. And it's going to assign-- because we're using the equal sign-- it's going to assign that value to that variable area.

Now, let's say we rebind radius to be something else. Radius i is bound to the value 2.2. But when we do this line, radius is equal to radius plus 1, we're going to take away the binding to 2.2. We're going to do this calculation. The new value is 3.2.

And we're going to rebind that value to that same variable. In memory, notice we're still going to have this value, 2.2, floating around. But we've lost the handle for it. There's no way to get it back. It's just in memory sitting there. At some point, it might get collected by what we call the garbage collector. In Python, And it'll retrieve these lost values, and it'll reuse them for new values, and things like that.

But radius now points to the new value. We can never get back 2.2. And that's it. The value of area-- notice, this is very important. The value of area did not change. And it did not change because these are all the instructions we told the computer to do.

We just told it to change radius to be radius plus 1. We never told it to recalculate the value of area. If I copied that line down here, then the value of area would change. But we never told it to do that. The computer only does what we tell it to do.

That's the last thing. Next lecture, we're going to talk about adding control flow to our programs, so how do you tell the computer to do one thing or another? All right.