PROFESSOR JIM ERICSON: OK, to work. A word of warning: fasten your seat belts. Or, another way of saying it is, I'm going to open up the fire hose a little bit today.

Last lecture, you might have thought this was a SHASS class, it's not like a philosophy class, and it was important to set the stage for what we're going to talk about, but we talked about very high level things. The notion of recipes, the notion of computation, why you want to do this, what you're going to learn.

Today we're going to dive into the nitty-gritty, the nuts and bolts of the basics of computation, and in particular, what I'm going to do today is, I'm going to talk about operators and operands, which we did a little bit real last time, in particular how to create expressions, I'm going to talk about statements as the key building blocks for writing code, and I'm going to introduce simple sets of programs, in particular I'm going to talk about branching, conditionals, and iteration. So, a lot to do. OK? So, let me jump straight to it.

At the end of last lecture, we started introducing some of the pieces you want to do. And I want to remind you of our goal. We're trying to describe processes. We want to have things that deduce new kinds of information. So we want to write programs to do that.

If we're going to write programs, we need at least two things: we need some representation for fundamental data. And we saw last time two examples of that. And the second thing we're going to need, is we're going to need a way to give instructions to the computer to manipulate that data. We need to give it a description of the recipe.

In terms of primitive data, what we saw were two kinds: Right? Numbers and strings. A little later on in the lecture we're going to introduce a third kind of value, but what we're going to see throughout the term is, no matter how complex a data structure we create, and we're going to create a variety of data structures, fundamentally all of them have their basis, their atomic level if you like, are going to be some combinations of numbers, of strings, and the third type, which are booleans, which I'm going to introduce a little later on in this lecture.

And that kind of makes sense right? Numbers are there to do numeric things, strings are our fundamental way of representing textual information. And so we're going to see how to combine those things as we go along.

Second thing we saw was, we saw that associated with every primitive value was a type. And these are kind of obvious, right? Strings are strings. For numbers, we had some variations; we had integers, we had floats. We'll

introduce a few more as we go along. But those types are important, because they tell us something about what we want to do when we want to put them together.

OK, but nonetheless, I want to stress we have both a value, yeah, and a type. All right. Once we have them, we want to start making combinations out of them. We want to put pieces together. And for that, we combine things in expressions. And what we saw as expressions are formed of operands and operators. And the simple things we did were the sort of things you'd expect from numerical things.

Now I want to stress one other nuance here. Which is, and we're going to do some examples of this, initially we just typed in expressions into the interpreter; that is, directly into Python. And as I suggested last time, the interpreter is actually a program inside of the machine that is basically following the rules we're describing here to deduce the value and print it up. And if we type directly into the interpreter, it essentially does an eval and a print. It evaluates, and it prints. Most of the time, we're going to be doing expressions inside of some piece of code, inside of a script, which is the Python word for program. In there, I want to make this distinction, this nuance: the evaluator is still going to be taking those expressions and using its rules to get a value, but it's not going to print them back out.

Why? Because typically, you're doing that to use it somewhere else in the program. It's going to be stored away in a variable. It's going to be stuck in a data structure. It's going to be used for a side effect. So, inside of code, or inside of a script, there's no print, unless we make it explicit. And that's a little bit down in the weeds, it's a detail, but one I want to stress. You need to, if you want something to be printed out inside your code, you need to tell the machine to do that.

OK. So let's do some simple examples. We've already seen somebody's. I just want to remind you, if I wanted to, for example, type in an expression like that, notice the syntactical form, it's an expression, a number, followed by an operand, followed by another expression. And of course I get out the value I'd like there.

Yes sir. Oh, you don't like leaning that far to the left? OK, if you're a Republican I'll be happy to shift this over a little bit. Wow, John, I got a laugh for a political joke, I'm in big trouble. That better? Oh damn, all right, I'll have to do it even more. OK, here we go, here we go, you see, I'm doing it down here, I can't see it, does that-- ah, I hear his sighs of relief, OK, good. There we go. Better. All right.

One of the other things we showed last time is that operators are overloaded. And this is where you heard John and I disagree. I don't happen to like this, but he thinks it's an ok thing. In a particular-- if we, whoa, we don't do that, we do this-- that is, give a combination of a number multiplication in a string, this will in fact give us back a new string with that many replicas, if you like, of the string concatenated together. All right? And if you want to do other things, for example, we can take two strings and add-- whoops, sorry-- and add them together, we will get

out, again, a concatenation of that string. And these will, we'll let you work through the variations, but these are the simple expressions we can use. Now, sometimes things get a little interesting. All right? What's the value of that expression? What do you think should happen if I evaluate that expression? Somebody with a hand up, so I can see it. What's going to happen?

STUDENT: [UNINTELLIGIBLE]

PROFESSOR JIM ERICSON: An error? Why?

STUDENT: [UNINTELLIGIBLE]

PROFESSOR JIM ERICSON: Great. OK. That means, let's check it. It certainly is. We bribe people. So I, ah, by the way, John's a Yankees fan, he throws like Johnny Damon, I'm a Red Sox fan, so we'll see if I, how about that? And I almost hit John along the way, great. My third right, exactly, what can I say? All right, so we're into bribing you as we go along here, and all right? You'll be badly overweight by the end of the term.

Right, it's a syntactic error, because it doesn't know how to deal with this. But there's an important thing going on here, if I in fact wanted to combine those into a string, I should have told the machine to do that, and I can do that, by explicitly saying, take that, which is a number, and convert it into a string, and then-- bleah, I keep doing that-- then add it to that string.

OK, so there's an important point here. We've got what's called type conversion. That is, if I want to combine two things together in a particular way, I need to make sure that I give it the kind of operand it expects. So STR, which I just typed up there, takes in parens, some input, and it converts it into a string, so that now I can use that where I was expecting a string. John.

PROFESSOR JOHN GUTTAG: You've got a static semantic error in your syntax.

PROFESSOR JIM ERICSON: Thank you. And I was going to come to that in a second, but thank you, John, for pointing it out.

All right. Why is it a static semantic error? The syntax is OK in the sense of, it is an operand, an operator, an operand, so syntactically it's OK. The semantics was what caused the problem, because the operator was expecting a particular kind of structure there.

There's a second thing going on here that I want to highlight, because it's really important. Yes indeed. OK, there we go. The second thing I want to highlight is, that what's going on, is that Python is doing some type checking. It caught the error, because it checked the types of the operands before it applied things, and it says, I'm going to

stop.

Now, you might have said, gee, why didn't it just assume that I wanted to in fact treat these as strings, and combine them together? Sounds like a reasonable thing to do. But it's a dangerous thing. Because in doing that, Python would then have a value that it could pass on into some other part of a computation, and if it wasn't what I wanted, I might be a long ways downstream in the computation before I actually hit some result that makes no sense. And tracing back where it came from can be really hard. So I actually want to have type checking as much as I can early on.

And in fact, under type checking, different languages sometimes fall on a spectrum from weak to strong typing. Which basically says, how much type checking do they do? Now, you're going to hear John and I go back and forth a lot, as I said I'm an old time-- well I'm certainly old time, but I'm also an old time Lisp programmer. I love Lisp, but Lisp is certainly in the category of a very weakly typed language. It does not check the types of its arguments at all. Python is, I wouldn't say completely strong, but it's much closer to the strong end of the spectrum. It's going to do a lot of type checking for you before it actually passes things back. Nonetheless, I'm also going to argue that it's probably not as strongly typed as we might like.

So, for example, there's an expression. Now, less than is just, if you haven't used it before, it's just the operator you'd expect, it's comparing two things and it's going to return either true or false depending on whether the first argument is less than the second argument. What's going to happen here? Again, I need a hand so I can know where to throw candy. I've also got on my reading glasses on, I can't see anything. Anybody. TAs don't count, they get their own candy. When it, yep.

STUDENT: [INAUDIBLE]

PROFESSOR JIM ERICSON: Good question. Sounds like a reasonable guess, right? How in the world am I going to compare a string to a number? So, see how good my aim is, ah, not bad. All right. A good quest-- sorry, a good thought, but in fact, son of a gun. Or as my younger son would say, fudge knuckle. Yeah. All right? So, what in the world's going on here? This is a place-- I don't know about you, John, I think this is actually really not good, because right, what this is doing is, it's allowing-- sorry, let me back up and say it-- it's got an overload on the less-than that allows you to compare basically the lexicographic ordering, or this sequence of ordering of symbols, including numbers inside of the machine. And this, in my mind, should have been an error. Why in the world would you want to compare that?

Just to give you an example of that, for instance, I can do the following: all right, the number four is less than the string three, whereas the string four, oops, is not less than the string three. And this is a place where it's comparing strings and numbers in a strange way.

So why am I showing you this? Partly to show you that it's kind of weird, but also to tell you that one of the things you want to do is exercise what I'm going to call some type discipline. Meaning, when you write code, you want to get into the habit of A, checking out operators or procedures to see what they do under different circumstances, either check them out or read the specifications of it, and two, when you write your own code, you want to be disciplined about what types of arguments or operands you apply to operators. Because this is something that could certainly have screwed you up if you didn't realize it did it, and you need to have that discipline to make sure it's doing the right thing.

OK. One of the other things you're going to see is that some of the operators have odd meanings. And again, I think we looked-- Yup?

STUDENT: So, the string A is less than three, is false because they're comparing like ASCII values?

PROFESSOR JIM ERICSON: Yes. I mean, I'm sorry. The answer is, I don't know if it's ASCII. John, do you know, are they doing ASCII encoding inside of here? I'm assuming so. Right. So, in case you didn't understand what the the question was, basically every symbol gets translated into a particular encoding, a string of bit, if you like, inside the machine, there's a particular one called ASCII, which is, if you like, an ordering of that, and that's what the machine's actually comparing inside of here, which is why in under ASCII encoding the numbers are going to appear after the characters, and you get the strange kind of thing going on.

All right. I want a couple of other things, just to quickly remind you, and one of them is, remember, the operators do look at the types, so division, for example nine divided by five is one, because this is integer division, that is, it's the largest number of integer multiples of five to go into nine, and there would be a remainder associated with it, which is in fact four. And again, you've got to be careful about how you use the operators.

Right, having done that, we can certainly get to more complicated things, so for example, suppose I look at that expression. Three plus four times five. All right. Now. There are two possible values here, I think. One is 23, the other's 35. Because this could be three plus four, times five, or it could be three, plus four times five. And of course, you know, when you look at code it doesn't pause in between them. But what I do? I just separated, do I do the addition first or do the multiplication first? Anybody know what happens In this case? Yeah, way up, oh God I'm going to have a hell of time throwing up there, way up at the back.

STUDENT: Standard order of operations, I guess take the multiplication first, and add the three.

PROFESSOR JIM ERICSON: Right. I'm going to try, if I don't make it, you know, just get somebody to pass back, whoa! I just hit somebody in the head. Thank you. Please pass it back to that guy. If you want candy, sit much

closer down, and that way we can film you as well as we go along. Right.

So the point is, there is something here called operator precedence, which is what the gentleman said. I'm not going to say much more about it, but basically what it says is, with all other things being equal, things like exponentiation are done before you do multiplication or division, which are done before you do things like addition and subtraction. And so, in fact, if I wanted the other version of it, in fact, if I do this right, it's going to give me 23 because it does the multiplication first, if I wanted the other version of it, I need to tell it that, by using, excuse me, parentheses. And in general, what I would say is, when in doubt, use parens. OK.

Now, that just gives us expressions. We can start having complex expressions, you can imagine we can have things are lots of parens, and all sorts of things in it. Yes, question.

STUDENT: What does it mean, the operator used, when you were calculating the remainder between nine and five?

PROFESSOR JIM ERICSON: It's the percent sign. If you can't read it, I guess I'm going to have to blow that font up, aren't I, next time around. Yeah, it's a percent, so this percent sign will give you the remainder.

OK. Second thing I need to do, though, is I need to, when I get those values, I want to hang on to them. I'd like to give them a name, so I can refer to them in other places. And so we saw that as well, the last piece we had here is the ability to create variables, which have their own values, and that's done using an assignment statement. So in particular, that is an assignment statement. It says, take the name x and create a binding for that name to the value of the sub-expression and in fact to do this, to stress a point, let's do that. It's not just a number, it's any expression. What Python will do, is it will evaluate that expression using the kinds of rules we talked about, and then it creates a binding for x to that value. And I want to stress this, we're going to come back to it later on in the term, so the way I'd like you to think about it for now, is that somewhere in the machine, there's a big space that contains all possible values. Right. It's a slight lie, it doesn't have all possible values, but you get the idea. It has, if you like, intellectually, all possible values. And when I create a binding, I'm taking a variable name, in this case x, stored somewhere in a table, and I'm creating a link or a pointer from that name to that value. This is a nuance. It's going to make a lot more sense later on when we introduce mutation into our language, but I want you to start thinking of it that way. Don't think of it as a specific box into which we're putting things; think of it as a link to a value.

I could have, for example, another assignment statement, and that creates a binding from y into that same value, and one of the things as a conservist I can do is, I could have a statement like, let z be bound to the value of x. And I said it deliberately that way. That statement says, get the value of x, which is this link, and give z a pointer to the same place. To the value, not to x. OK, and we'll just plant that idea, we're going to come back to later on, as

we carry on.

OK. So if we have variables, one of the questions we can ask is, what's the type of the variable. And the answer is, it inherits it from its value. OK. Yes. So if somewhere in my code, I have that statement, that assignment statement, x now is a variable whose value is an integer. Unfortunately, at least in my mind, in Python, these variable bindings are dynamic, or the type, rather, is dynamic. Meaning, it changes depending on what the current value is. Or said a different way, if somewhere later on in the program I do this, x now has changed its type from INT to string. Now why should you care? OK, my view is, I don't like it. Especially in the presence of operator overload. Because I might have written some code in which I'm expecting that particular variable to have an integer value. If somewhere later on in the code it shifts to string, I might not be manipulating that and getting actual values out, but not what I wanted, and it's going to be really hard for me to chase it back. So one of the things I would like to suggest is that you develop some good style here, and in particular, don't change types arbitrarily. I can't spell today. Meaning, sometimes you need to do this, but in general there's-- at least in my view and I don't, John, would you agree?-- you just don't want to do this. You don't want to make those changes. It just leads to trouble down the road.

OK. Now, last thing about variables, and then we're going to start pushing on this, is where can you use them? And the answer is, you can use a variable anywhere you can use the value. So, any place it's legal to use the value.

OK. Now. This is just sort of bringing us back up to speed and adding a few more details in. What we really want to do now though is start using this stuff. So, operands. Let us take expressions, get values out, we can store them away in places, but ultimately we want to do something with them, so we need to now start talking about what are the things we can do inside of Python, or any programming language, to manipulate them. And for that, we're going to have statements.

Statements are basically, if you want to think about it, legal, and I was about to use the word expression except I've misused that elsewhere, so legal commands that Python can interpret. You've already seen a couple of them. Print, assignment, certainly two obvious statements, they're commands to do something. Assignment is binding a name to a value, print is saying put it back out in the screen. Obviously if you have print as a way of putting things out, we expect to have ways of getting input in, we're going to see an example that in the second. And as we go through the next few lectures, we're going to add in more and more of these statements.

But let's look at what we could do with this, OK? And to do this, I'm going to use some code that I've already typed in. So I'm hoping you can read that, and it's also in your handout. This is a little file I created, all right, and I'm going to start with a sequence of these things and walk them along, again I invite you to put comments on that

handout so that you can follow what we're going to do. All right?

So let's look at the first part of this. Right now, this is just a text file. OK. And I've highlighted in blue up there one of the pieces I'm going to start with. And what do I have? I have a sequence of commands; I've got an assignment statement, I've got another assignment statement, I've got a print statement, I've got an input statement, which we'll come back to in a second. And I want to basically try and use these things to do something with them.

Second thing I want to note is, the little hash mark or the pound sign, that's identifying a comment. So what's a comment? It's words to you, or to the reader of the code, that are telling you what's going on inside of this code. OK?

Now, these comments, frankly, are brain-damaged, or computationally challenged if you prefer. Meaning, why in the world do I have to tell the reader that I'm binding x to the value three? All right? I'm putting them in there to make a point. In general, good programming style says you put in comments that are going to be valuable in helping you as a reader understand what's going on inside of the code. It could be, what's the intuition behind this piece of code. It could be, preconditions I want to have on input. It could be, explanations of specific things you're doing. But you need to have those comments there.

Now, this becomes a little bit of one of those motherhood and apple pie kinds of lectures. You know, your mother always told you to eat brussels sprouts because it was good for you. Well this is a brussels sprouts comment. everybody goes yeah, yeah, yeah, comments, of course. Of course we're going to do comments. And they never do. So my challenge to you, and I know Professor Guttag can do this, my challenge to you is, a year from now, come back and look at code you wrote here. Can you still understand what it was you were trying to do? I don't know, John, if you'd agree, right? If you can read the code a year later, even code you wrote yourself, it's a good sign that you put good comments in, right?

Second good piece of style here is choice of variable names. These are lousy. Deliberately. OK? I'm just using simple things like x and y and z because I want to make it, just get through the lecture if you like. But in general, the choice of variable name is a great way of commenting your code. Use variable names that make sense. That little problem set zero that you did. You read in a couple of values, you probably stored them away. My bet is, you used simple names like x and y. A much better name would have been firstname, lastname, as the name of the variable to tell you what you were trying to capture there.

OK. The other piece I want to say about variable names is, once I have that choice of variable name, I can use it, but in fact there are a few things that I can't use in terms of variable names. So, these are an important way of documenting, but there're some things excluded. And in particular, there are some key words that Python is going to use that have to be excluded. Let me highlight that. As I said, right now that's just text file. I'm going to save this

away-- yeah, not that way, I'm going to save this away-- with the subscript, or the suffix rather, py to make it a Python file. Yeah, and I know it's already there but I'm going to do it, and I get some wonderful colors. But these are important, OK? So notice what I have up there now. Comments appear in red. I can see those. There's a keyword, which I'm going to highlight right up here, print, which is in, I don't know what that color is, orange? There's a function in purple, there's a string in green, and in black I have the assignment statements.

That print is a keyword. It's a command to Python to do something. As a consequence, I can't use it as a variable name. All right, think about it for a second. If I wanted to use print as a variable name, how do I get the system to decide gee, do I want print as a value for something, or do I want print as a command? So there's a sequence of these that are blocked out, and I-- John, I think there are what, twenty-eight? Something like that, TAs, is that right? Twenty-eight keywords that are blocked? We'll find them as we go along--

OK. Having done this now, I can simply go ahead and run this, and in fact if I go up here to run, you'll see I've got both an option to check the module, though in this case I'm just going to run it. Oh, notice what happened. It ran through that sequence of instructions, in particular it bound x to the value three, and then it took x times x, got the value of x multipied by x, which of course is nine, bound that to the value of x, and then it printed out the value, and now it's sitting here waiting for an input. You notice what it did, it printed out that little, right up here I'd said enter a number and that's what it's printed out, so I can enter a number and it prints it out. Great.

Let's run it again. Actually for that, I can just use, if I'm lucky, function F5, which didn't work, so let me try it again., here we go. We're going to run that module. OK.

Whoa. What happened? I said enter a number. I didn't. I gave it a string. And it still took it. And printed it up. Well, this is one of the places where I want to come back to that highlighting of what do things do? Even though my statement said enter a number, in particular, raw input here simply takes in a set of characters and treats it as a string. And then prints it back out. So if in fact I wanted to make sure this was a number, I should have done something like either try and convert it to a number, which of course failed here, or put in a check to say where it is. So it's a way of reminding you, I've got to be careful about the types of things that I put in.

OK. This is still boring, so let's step on the accelerator. What I have now is the following: I can write expressions, do combinations of things to get out values, I can store them away, I can print them up. But literally all I can do at this stage is write what we would call a straight-line program, that is, a program in which we execute in which we execute the sequence of instructions one by one. Simply walk down that list. That's what we just did there, right? We just walked through that list. This is boring. In fact, you can do some nice things to prove what is the class of functions you can compute with straight-line programs, and what you'd see if you did that is, it's not particularly interesting.

OK. Let's go back and think about our recipes. What we use as our motivation here. Even in real recipes, you have things like, if needed, add sugar. That's a decision. That's a conditional. That's a branch. That says, if something is true, do something. Otherwise, do something different. So to really add to this, we need to have branching programs. What I mean by that is, a branching program is something that can change the order of instructions based on some test. And that test is usually a value of a variable. OK. And these get a whole lot more interesting.

So let's look at a little example, and this is going to, excuse me, both allow us introduce the syntax as well as what we want to have as the flow of control inside of here. So let me go back up here, and I'm going to comment out that region, and let's uncomment this region. I want to write a little piece of code. It's going to print out even or odd, depending on whether the value I put in, which is x in this case, is even or odd.

Think about that. That says, if this thing has some particular value, I want to do one thing; otherwise, I want to do something different. And let's look at the syntax of this. This is the first of the conditionals that we're going to see. Notice the format. I'm going to go up there. The first statement right here, that's just an assignment statement, I'm giving some value to x. We could make it something different. And then, notice the structure here. The next three statements.

First of all, IF is a keyword. which makes sense. It is followed, as you can see there, by some expression, followed by a colon. And in fact, that colon is important, so let me stress this over here. The colon is important It's defining the beginning of a block of instructions. Yes sir.

STUDENT: [UNINTELLIGIBLE]

PROFESSOR JIM ERICSON: Based on a test. Usually the value of a variable.

OK, so let me go back to where I am. I'm looking at that piece of code. What that colon is saying is, I'm about to begin a sequence of instructions that I want to treat as a block. So it identifies a block of instructions. It's, and in particular, the colon is the start, and the carriage return is the end.

Now what in the world does that mean? I'm doing a lot of words here, let me try and say this a little bit better. That code says the following: the IF says, I've got an expression, I'm going to evaluate it. If that value is true, I want to do a set of things. And that set of things is identified by the sequence of commands that are indented in, this one right here, following the colon but before I get back to the same place in terms of the indentation. If that test is not true, I want to skip this instruction, and there's a second keyword else, followed by a colon, and that tells me the thing I want to do in the case that it's false.

So in fact if I run this, ah, and save it, and it prints out odd. So, what happened here? Well, let's look at the code. Right? x is initially bound to fifteen. I get to the IF. The IF says, evaluate that next expression. In that next expression, I'm actually taking advantage of the fact that I'm doing integer multiplication and division here. Right, that divide is, if x is an integer and two is an integer, what's it going to do? If x was even, x divided by two is going to be actually the half of x, right? If x is odd, that integer division is going to give me the number of multiples of two, that go into x, plus a remainder, which I'm going to throw away. In either case, I take that value and multiply back by two, if it was even I get back the original number, if it was odd, I'm not going to get back the original number, so I can just check to see if they're the same.

OK, so a little nuance that I'm using there. So, the first thing that IF does, bleah that IF says is, evaluate that expression and if it's true, do the next thing, the thing after the colon. In this case it's not true, so it's going to skip down and evaluate the thing printed up the odd.

OK. What-- yes.

STUDENT: [INAUDIBLE]

PROFESSOR JIM ERICSON: Thank you. I was hoping somebody would ask that question. The question was, if you didn't hear, why do I have two equal signs? It's like I'm doing this, right? Anybody have a se--%uFFFD why don't I just use an equal sign? I want to know if something's equal to something. Yeah.

STUDENT: [INAUDIBLE]

PROFESSOR JIM ERICSON: Absolutely. The equal sign is going to bind-- Nice catch. John, this is so much fun, throwing candy. I've got to say, we've got to do this more often-- Right. Let me, let me get to the point. What does an equal sign do? It is an assignment. It says, take this thing on the left and use it as a name to bind to the value on the right. It's not what I want here. Having already chosen to use equal as an assignment, I need something else to do comparison. And that's why I use double equals. Those two equal signs are saying, is this thing equal to, in value, the thing on the other side?

OK. Now, having done that, again I want to stress this idea and I'm going to write it out one more time, that there's a particular format here. So we have if, and that is followed by, I'm going to use angle braces here just to indicates something goes in here, some test followed by a colon. That is followed by a block of instructions. And we have an ELSE, followed by a colon in some other block of instructions. And I want you to get used to this, that colon is important, it identifies the start, and then the set of indented things identify all the things at the same level, and when we reset back to the previous level, that's when we go back to where we were.

OK. Now, that's a nice simple little test, let's look at a couple of other examples to get a sense of what this will do.

OK, let me comment this out, and let's look at this next little piece of code. All right. I'm binding a z to be some value, and then I'm going to run this. Well, let's just run it and see what it does.

Nothing. OK, so why? Well, let's look at it. I'm doing a test there to say, if the string x is less than the value of b, and x does not appear before b as strings, then I was going to do, oh, a couple of things, because they're at the same block level. Given that that wasn't true, it did nothing.

Now, wait a minute, you say, where's the ELSE clause? And the answer is, I don't need one. All right, if this is purely a test of, if this is true do this otherwise I don't care, I don't need the ELSE clause in there to identify it. All right?

Second thing I want to look at is, suppose I compare that the one below it. Oops, that I don't want to do. Comment that out, and let's uncomment this. Yeah, I've still got a binding for z and I'm giving it the same test, but notice now I've got the two same commands but they have different indentation. In this case, in fact I do get a different behavior.

Why? Because that block identifies a set of things that I'm going to do if the test is true. If the test was not true, notice that that last command for print Mon is now back at the same level as the IF, so what this says is the IF does the test, having done the test, it decides I'm not going to do anything in the block below it, I'm going to skip down therefore to the next instruction at the same level as the IF, which gets me to the second print statement.

OK. So now we're seeing some of these variations, let's see what else can we do here. So let me just to try something a little more interesting, and then we'll get to writing some simple programs. So I'm going to comment those out, and let's go down to this piece of code, and uncomment it. Ooh yes, that was brilliant. Let's try this again. And uncomment that, and uncomment it again. Right, so here's a little piece of code that's going to print out the smallest value of three. And notice what this is showing is that the IFs can be nested. All right, it's so if I looked at it, it's going to say that IF x is y-- sorry, IF x is less than y, THEN check to see IF x is less than z, and if that's true, print out x is the smallest. And notice the structure of it, if it's not true I'm going to go to that next ELSE, and print out that z is smallest. If the first test wasn't true, I'm going to skip that whole block and just go down and print out that y was smallest. So notice the nesting, I can flow my way through how those tests are actually going to take place.

All right, so let's run this and see what happens. Great. y is smallest. OK. Is that code correct? Is that a tentative hand back there? Yeah.

STUDENT: Let me compare y to [INAUDIBLE]

PROFESSOR JIM ERICSON: Yeah, it's not doing all of the comparisons. All right, and let's just check this out, because I want to make a point of this, let's go back and do the following. Let's take y, change it to thirteen, let's run it, hmm.

So what did I miss here? Two important points. First one, when I write a piece of code, especially code that has branches in it, when I design test cases for that piece of code, I should try and have a specific test case for each possible path through the code. And by just doing that, I just spotted, there's a bug here. And the bug was in my thinking, I did not look for all of the tests. So the way I can fix that, is, let me comment that out, and keep doing that, comment that out, let's uncomment this, notice the structure here. I now have multiple tests. So actually, let's just run it and then we'll talk about what it does.

I run this, yeah, I have a syntax error, yes indeed, because I forgot to comment that one out, all right, and cue, we'll try it again. Ah-ha!

And let's quickly look at the structure of this. This now has, gee, a funny thing, it says IF x is less than y AND x is less than z, then do something. And then it has a strange thing called ELIF, which is simply short for else/if in a second test. So the way to think about this in terms of flow is, it starts with that if and it says, check both of those things.

And that fact that both of those things is the fact that we're using Boolean combination here. It is to say, we can take any logical expressions, combine them together with AND, OR, or NOT to make a complex expression, and use the value of that expression as my test. And that's literally what I've done there, right, I've got x less than y, that's a test, it returns a Boolean, which by the way is the other type, at least I would include here, it has only two values, which are true and false, and what that code says, if x is less than y, AND, logically, whatever I put up there, x is less than z, then the combination is true, and therefore I'm going to do something.

So AND is if both arguments are true it's true, OR is if either argument it's true it's true, NOT is if the argument is not true it's true, and then the last piece, as I said is, I can now have a sequence of things I want to do. So if this is true do something else, otherwise test and see if this is true, do something else, as many as I like followed by the end. And ELSE, it says, here's what I want to do.

OK. Now. Having added this in, I have branching instructions. I have simple branching programs. These are still awfully simple. OK? And they're awfully simple because, all I can do now, is decide whether to execute some piece of code or another. Said a different way, in the case of the straight-line programs, how long would it take to run a program? Well, basically, however many instructions I have, because I've got to do each one in order. With simple branching, how long is it going to take to run a piece of code? Well at most, I'm going to execute each instruction once. All right? Because the IFs are saying, if it's true do this, otherwise skip on it.

Therefore, for simple branching programs, the length of time, the complexity the code, is what we would call constant. That is, it's at most the length of the actual number of instructions. It doesn't depend on the input. Real simple programs.

Let's take another simple example. Suppose I want to compute the average age of all the MIT faculty. There's about a thousand of us. However I do that, I know that should inherently take more time than it takes to compute the average age of all the EECS faculty. There's only 125 of us. And that should take more time than what it takes to compute the average of John's and my ages, instructors in 600, because there's only two of us.

All right, those pieces of code inherently involved something that does depend on the size of the input, or on the particular input. And that is a preface to an idea of computational complexity we're going to come back to. One of the things we want to help you do is identify the different classes of algorithms, what their costs are, and how you can map problems into the most efficient class to do the computation.

OK. Now. Think for a second about computing the average age of the faculty. You can already kind of see what I want to do. I, somehow if, I want to walk through some sequence of data structures, gathering up or doing the same thing, adding ages in until I get a total age and then divide by the number faculty.

How do I write a piece of code for that? Well, let's go back up to our original starting point of recipes. And I'm sure you don't remember, but one of the things I had in my recipe, is beat egg whites until stiff. OK. That until is an important word. It's actually defining a test. Let me rephrase it into garbled English that'll lead more naturally into what I want to do. While the egg whites are not stiff, beat them. That is a different kind of structure. It has a test in it, which is that while, while something is true, do something, but I want to keep doing it.

And so for that, we need to add one last thing. Which is iteration. Or loops. We're going to see variations of this, we're going to see a variation of it called recursion, a little later on, but for now we're just going to talk about how do we do iterations. And I want to show you an example of this, to lead to both the syntax, and to the semantics. And let me comment that out, and let's go to this one.

All right. What does this piece of code say? Not, what does it do, but what does it say. Well, the first three statements are just assignment statements. I'm binding x, y, and iters left to some values. And then notice the structure, I got a keyword WHILE, there's that color identifying it, and in parentheses I have a test. I'm expecting the value that test to be a Boolean. Followed by a colon. The colon's identifying a block of code. And what this is saying is, gee. Check to see if the variable iters left has a value greater than zero. If it does, then execute each of the instructions in that block.

So I'm going to have an assignment of y, I'm going to have an assignment of iters left, I've got a comment that I had borrowed in order to do some debugging, and then what do I want it to do? I want it to go back around to the test. Once again, say, is that test true? If it is true, execute the sequence of instructions. So in fact we can block this out and see what it does. If I make a little chart here, I've got x, y, and iters left. x starts off as I think I set it up as, here I can't even read it, is x is three, y is zero, iters left is three. I can hand-simulate it. It says, is the value of iters left greater than zero? Yes it is. So, execute those two instructions. It says, take value of y and value of x, add them together, and create that as the new value of y. All right. That's the assigned statement. It says, take iters left, subtract one from it, and bind that as the new value of iters left. Having reached the end of the block, go back up and check the test. Is iters left greater than zero? Yes it is. So, evaluate the same sequence of instructions again. y plus x is six, that's my new value of y, two minus one is one, that's my new value of iters left, go back up. Is iters left greater than zero? Yes it is. So once more, thank God I didn't take 47 as an example, x plus y, subtract one from iters left, go back up to the test. Is iters left's value greater than zero? No, it is not. Therefore, skip the rest of that block of code and go to the next instruction, which is, ah, print out y.

In fact, if we test this, son of a gun. Got a simple square procedure,. Right, It's just squaring an integer, is what it's doing.

But notice the structure. Now I have the ability to create a loop, that is, to reuse the same pieces of code over and over again as I go around. And this adds, now, a lot of power to the kinds of code I can write.

Notice some other things I want to highlight on this. All right? The first one is, that test has to involve-- shouldn't have to, but almost always is going to involve-- the value of some variable. What if I don't change the value of that variable inside of the code? Or, another way of saying it is, what if I did this?

Comment it up. What happens if I run this sucker?

STUDENT: [INAUDIBLE]

PROFESSOR JIM ERICSON: Yeah. It'll go forever. Absolutely, right? It's going to loop into an infinite loop-- I think I can hit this close, ah, no I can't, boy what a terrible aim-- All right, what has [UNINTELLIGIBLE PHRASE] try again, the point I'm trying to make here-- thank God we're at the end of this lecture, my tongue is falling apart-- the point I'm trying to make is, that test needs to involve some loop variable that is changing. Otherwise the test is always going to be true, we're going to go off here, but this would loop forever if I did that.

All right. Second question: or maybe a better way of saying this, and the general format you're likely to see here is, a test involving a variable name, which must be initialized outside of the loop, and which interior to the loop gets changed, so that the test is going to change. Here's the second question. What value of inputs, what values of x

will this run correctly for? Probably should be integers, right? Otherwise, this is going to be doing something strange, but all integers?

All right, suppose I do this. It's my last example. Yeah, how about that, right? We knew this was trying to do squaring, so intellectually we know we can square -4, it ought to be 16, but what happens here? Double fudge knuckle. All right? It's going to run through the loop, accumulating the answers, but because I'm subtracting, it's just going to keep making x more and more negative as it goes along, again it's off into an infinite loop. Which is a way of reminding you that I need to think as I write the code about what are my expectations from the input, and how might I enforce them. In this case, I probably want to make sure I use absolute value of x before I carry it on. Yes ma'am.

STUDENT: [UNINTELLIGIBLE]

PROFESSOR JIM ERICSON: You're absolutely right, because I bind iters left to, um, yeah. Yes. Thank you. Boy, two candy for you. You caught me making an error. Yes. The point is, it's not going to work, and she caught both of them, impressive, it's not going to work because iters left is already negative, it's just going to skip the whole loop, and I'm in trouble, so thank you for catching that.

All right. I was going to do one more example, but I've run you up to the end of the time. I'll leave the example on the handout, it shows you another version that we'll come back to next time. The key thing to notice is, I now have the ability to create iterations, which extends well I can do. And we'll see you next time.