MIT OpenCourseWare
http://ocw.mit.edu

6.00 Introduction to Computer Science and Programming, Fall 2008
Transcript – Lecture 3

PROFESSOR ERIC GRIMSON: All right, I'm going to start today by talking about, so what have we been doing? What have we actually done over the last few lectures? And I want to suggest that what we've done is, we've outlined a lot of the basic elements of programming. A lot of the basic elements we're going to need to write code. And I want to just highlight it for you because we're going to come back and look at it.

So I'm going to suggest that we've looked at three different kinds of things. We've talked about data, we've talked about operations, and we've talked about commands or statements. All right?

Data's what we expect. It's our way of representing fundamentally the kinds of information we want to move around. And here, I'm going to suggest we've seen numbers, we've seen strings, and I'm going to add Booleans here as well. They're a third kind of value that we saw when we started talking about conditions.

We saw, associated with that primitive data, we have ways of taking data in and creating new kinds of data out, or new versions of data out, so we have operations. Things like addition and multiplication, which we saw not only apply to numbers, but we can use them on things like strings and we're going to come back to them again. Can't use them on Booleans, they have a different set of things. They do things like AND, and OR. And of course there's a bunch of other ones in there, I'm not going to put them all up, but we're building up a little collection, if you like, of those operations.

And then the main thing we've done is, we've talked about commands. So I'm going to suggest we've seen now four different things. We've seen assignment, how to bind a name to a value. We've seen input and output. Print for output, for example, and raw input for input. We've seen conditionals, or said another way, branches, ways of changing the flow of control through that sequence of instructions we're building up. And the last thing we added were loop mechanisms. And here we saw, wow. It's the first example we've seen.

So what've we done so far? Now, interestingly, this set of instructions was actually quite powerful, and we're going to come back to that later on, in terms of what we can do with it, but what we've really done is, given that basis, we're now talking about, how do we write common patterns of code, how do we write things that solve particular kinds of problems. So what I want you to do, is to keep in mind, those are the bases, we ought to be able to do a lot with that bases, but what we're really

interested in is not filling out a whole bunch of other things in here, but how do we put them together into common templates. And we're going to do that today.

Second thing we've been doing, I want to highlight for you is, we've along the way, mostly just verbally rather than writing it down, but we've been talking about good style. Good programming style. All right? Things that we ought to do, as you put these pieces together, in order to give you really good code. And you should be collecting those together.

Give you some examples. What have we talked about? We've talked about things like using comments to highlight what you're doing in the code, to make it easier to debug. We talked about type discipline, the notion that you should check the types of operands before you apply operators to them, to make sure that they're what the code is expecting. We talked about descriptive use of good variable names, as a way, in essence, of documenting your code. The fourth one we talked about was this idea of testing all possible branches through a piece of code, if it's got conditionals in it, to make sure that every possible input is going to give you an output that you actually want to see.

So, you know, you can start writing your own, kind of, Miss Manners book, if you like, I mean, are what are good programming, you know-- I wonder what you'd call them, John, good programming hygiene? Good programming style? Good programming practices?-- Things that you want to do to write good code.

OK. What we're going to do today is, we're going to start now building up, beyond just these pieces, although they're valuable, to start creating two things: one, common patterns of code that tackle certain classes of problems, and secondly we're going to talk about tools you can use to help understand those pieces of things.

OK. So last time around, we talked about, or introduced if you like, iterative programs. And I want to generalize that for a second, because we're going to come back and use this a lot. And I want to do a very high-level description of what goes into an iterative program, or how I would think about this, all right? And I know if John disagrees with me he'll tell me, but this is my way of thinking about it.

If I want to try and decide how to tackle a problem in an iterative matter, here the steps I'm going to go through. First thing I'm going to do, is I'm going to choose a variable that's going to count. What I meant-- what in the world do I mean by that? I'm thinking about a problem, I'm going to show you an example in a second, first thing I'm going to do is say, what is the thing that's going to change every time I run through the same set of code? What is counting my way through this process?

Now I'm putting count in double quotes, not to make it a string, but to say, this is count generically. It could be counting one by one through the integers, it could also be taking a collection of data and going through them one by one. It could be doing counting in some other mechanism. But what's the variable I want to use?

Second thing I do, I need to initialize it. And I need to initialize it outside of the loop. That is, where do I want to start? And I need to make sure I have a command that sets that up.

The third thing I'm going to do, is I need to set up the right end test. How do I know when I'm done with the loop? And obviously, that ought to involve the variable in

some way, or it's not going to make a lot of sense, so this includes the variable, since that's the thing that's changing.

All right. The fourth thing I'm going to do, is I'm going to then construct the block of code. And I want to remind you, that block of code is a set of instructions, the same set of instructions that are going to be done each time through the loop. All that's going to change, is the value the variable or the value of some data structures. And remind you that inside of here, I'd better be changing the variable. All right, if that variable that's counting is not changing, I'm going to be stuck in an infinite loop, so I ought to [UNINTELLIGIBLE PHRASE] that , right, expect somewhere in there, a change of that variable. All right? And then the last thing I want to do, is just decide, you know, what do I do when I'm done.

OK. I know. It looks boring. But it's a structure of the things I want to think about when I go through trying to take a problem and mapping it into a iterative program. Those are the things I want to see if I go along.

All right. So let me give you an example. I'm given an integer that's a perfect square, and I want to write a little piece of code that's going to find the square root of it. All right, so I'm cheating a little, I know it's a perfect square, somebody's given it to me, we'll come back in a second to generalizing it, so what would the steps be that I'd use to walk through it?

Well if you think about these steps, here's an easy way to do it. Let's start at 1. Let's call x the thing I'm trying to find the square root of. Let's start at 1. Square it. If it's not greater than x, take 2. Square it. If it's not greater than x, take 3. Square it. And keep going, until the square of one of those integers is greater than or equal to-- sorry, just greater than x. OK, why am I doing that? When I get greater than x, I've gone past the place where I want to be. And obviously, when I get to something whose square is equal to x, I've got the answer I want, and I kick it out.

So who knows what I've done? I've identified the thing I'm going to use to count, something some variable is going to just count the integers, I've identified the end test, which is when that square is bigger than the thing I'm looking for, I've identified basically what I want to do inside the loop, which is simply keep changing that variable, and I didn't say what I want to do when I'm done, essentially print out the answer.

OK, so how can I code this up? Well, you might think, let's just jump in and write some code, I don't want to quite do that though, because I want to show you another tool that's valuable for thinking about how to structure the code, and that is a something called a flow chart. Now. People of Professor Guttag's and my age, unfortunately remember flow charts back-- as they say, on the Simpsons, back in the day, back in the 1960's, John, right?-- really good programmers had these wonderful little plastic stencils, I tried to find one, I couldn't find it It's a little stencil with little cut-out shapes on it, that you used to draw flow charts, I'm going to show you in a second, and you tucked it right in here next to your pocket protector with all your pens in it, you know, so, your belt was also about this high, and your glasses were this thick, you know, we have a few of those nerds left, we mostly keep them in the museum, but that was what you did with the flow chart.

Despite making a bad joke about it, we're going to do the same thing here. We're going to do the same thing here, we're going to chart out a little bit of what should

go into actually making this thing work. So here's a simple flow chart that I'm going to use to capture what I just described. And I'm going to, again, I'm actually going to do it the way they used to do it, and draw a rectangle with rounded corners, that's my starting point, and then what did I say to do? I said I need to identify a variable, I'm going to give it a name, let's just call ANS, for answer, and I need to initialize it, so I'm going to come down, and in a square box, I'm going to initialize ANS to 0.

And now I want to run through the loop. What's the first thing I do in a loop? I test an end test. So the flow chart says, and the tradition was to do this in a diamond shape, I'm going to check if ANS times ANS-- oh, which way did I want to do this-- is less than or equal to x. Now that's a test. There are two possibilities. If the answer is yes, then I'm still looking for the answer, what do I want to do? Well, I don't have to do anything other than change the counter. So I'm going to go to ANS is ANS plus 1, and I'm going to do it again. Eventually, if I've done this right, that test is no-- and I wonderfully ran out of room here-- in which case, I'm going to go to a print statement, which was always done in a trapezoid, and print out ANS. I should have put a box below it that says stop.

OK? Wow. And notice what I got here. Actually, this is a useful tool for visualizing how I'm trying to put it together, because it lets me see where the loop is, right there, it lets me see the end test, it lets me make sure that I'm in fact initializing the variable and I'm checking the right things as I go along. And the idea of this flow chart is, if you start, you know, a little ball bearing here, it's going to roll down, setting up an assignment statement, and then, depending on here, it's like there's a pair of flippers in there, it does the test, it sets the ball this way to change it to ANS plus 1, and comes back around, eventually it's going to drop through and print out the answer.

The reason I'm going to show you this flow chart, I'm going to do one other example in a second, but I want to show you a comparison. Remember last time, we wrote this simple piece of code to print out even or odd. If, you know, x, it was in fact, even or odd. So let me show you what a flow chart for that would look like, because I want to make a comparison point here.

If I were to do a flow chart for that one, I'd do the following. It reminds you, that the test here was, we took x if that's what we were looking for, it did integer division by 2, multiplied it by 2, and we check to see if that was the same as x. If the answer is yes, then we did a print of even. If the answer was no, we did a print of odd, and we then carried on. Again, wow.

But there's an important point here. Remember last time, I said that there's different kinds of complexity in our code, and I suggested for simple branching programs, the amount of time it takes to run that program is, in essence, bounded by the number of instructions, because you only execute each instruction at most once. It didn't depend on the size of the input. And you can see that there.

I start off, either I take this path and carry on, or I take that path and carry on, but each box, if you like, gets touched exactly once.

On the other hand, look at this one. This depends now on the size of x. All right? Because what am I going to do? I'm going to come down and say, is ANS squared less than or equal to x? If it is, I'm going to go around, and execute that statement, check it again, and go around and execute that. So I'm going to cycle around that

loop there enough times to get to the answer, and that number of times is going to depend on the input, so as I change the input, I'm going to change the complexity of the code.

Now this happens to be what we would call a linear process, because the number of times I go around the loop is directly related to the size of the argument. If I double the argument, I'm going to double the number of times I go around the loop. If I increase it by five, I'm going to increase by five the number of times I go around the loop.

We'll see later on, there are classes of computation that are inherently much more complex. We hate them, because they're costly, but they're sometimes inherently that way. But you can see the comparison between these two.

OK. Now, having done that, let's build this code. Yeah, if my machine will come back up, there we go. So, I'm going to now go ahead and write a little piece of code, and I put it here and I hope you can actually see these better this time, let me uncomment that region. All right. So, there's basically an encapsulation of that code, right? It says-- what, look at this, where am I, right here-- I've got some value for x initially, I'm going to set ANS to 0, just like there, and there's my loop, there's the test, which is right like that, is ANS squared less than or equal to x, if it is, there's the block corresponding to the loop, change ANS, and eventually when I'm done with all this thing, I'm just going to print ANS out.

OK. All right, let me show you one other tool that I want to use. Which is, I've written that piece of code, I ought to check it. Well, I could just run it, but another useful thing to do is, I'm, especially as I want to debug these things, is to simulate that code. And I'm going to do this because, as Professor Guttag noticed to me, students seem reluctant to do this. I guess it's not macho enough, John, to just, you know, you know, go off and do things by hand, you ought to just run them, but it's a valuable tool to get into, so let me do that here.

STUDENT: [UNINTELLIGIBLE]

PROFESSOR ERIC GRIMSON: I'm doing such a great job. I've got to say, when my, I've got two sons, now aged eighteen and twenty, they used to think I had the coolest job in the world because I came home covered in chalk. Now they have a different opinion that you can probably figure out.

All right. Simulate the code. What I mean by that is, pick a simple set of values, and let's walk through it to see what happens. And this is useful because it's going to allow me to A, make sure that I've got something that's going to terminate, it's going to let me make sure that in fact I'm doing the right kinds of updates. I could do this, by the way, by running the code and putting print statements in various places as well, but the hand simulation is valuable, so let me just start it.

What do I have here? I need the variable, ANS, I need x, and I need ANS times ANS, ANS times ANS. Right. Those are the three things that are involved in this computation. and I pick something reasonably simple. The ANS starts at 0. I set up x, I think, to be 16 there. So what does the loop say? I can either look at my flow chart, or I can look at the code. If I look at the flow chart, it says, I'm at this point. Look at ANS squared. Is it less than or equal to-- sorry, first of all, ANS squared is 0, is it less than or equal to x, yes. So what do I do? Change ANS. X doesn't change.

Back around to the test. What's ANS squared? It's 1. Is it less than or equal to 16? Sure. Run the loop again. ANS becomes 2. X stays 16. ANS squared is 4. Is that less than or equal to 16? Yes. Aren't you glad I didn't pick x equals 500? All right. ANS goes up by 0. ANS squared is nine. Still less than or equal to 16. ANS goes to 4. X stays the same. 4 squared is 16. Is 16 less than or equal to 16? Yes. So ANS goes to five. ANS squared becomes 25. Ah! That is now no longer true here, so I print out 5. Right. Sure. Square root of 16 is 5. It's Bush economics. OK? I know. I'm not supposed to make bad jokes like that.

What happened? Yeah.

STUDENT: It doesn't stop at the right place.

PROFESSOR ERIC GRIMSON: It doesn't stop at the right place. Thank you. Exactly. Right? My bug here is right there. Ah, let me find my cursor. I probably want that. Right? I want less than, rather than less than or equal to. This is an easy bug to come up with. But imagine, if you don't do the test, you're going to get answers that don't make any sense. And in fact, if we just go ahead and run this now, hopefully we get out-- oops, sorry, I'm going to have to change this quickly, I still have some things uncommented at the bottom, yeah, there they are, I don't think we need that yet, all right, we will comment those out.

OK. So. Why did I do it? It's a simple example, I agree, but notice what I just did. It allowed me to highlight, is the code doing the right thing? I spotted an error here, I could have spotted it by running it on different test sets, and using prints things, another way of doing it, but this idea of at least simulating it on simple examples lets you check a couple of important questions.

And in fact, now let me ask those two questions about this piece of code. First question is, for what values of integers-- we're going to assume integers-- but for what values of x does this code terminate? And the second question is, for what values of x does it give me back the right answer?

All right, first question. What values of x does it terminate? Again, assume x is an integer. Well, break it down into pieces. Suppose x is positive. Does it terminate? Sure. All right? Because ANS starts out as 0, so ANS squared is 0, and each time through the loop, ANS is increasing. That means, at some point, in some finite number of steps, ANS squared has got to get bigger than x if x is positive. So for positive integers, it terminates. And it probably, I think we can deduce, returns the right answer here.

Right. X is negative. X is -16. Does this code terminate? Boy, I feel like Arnold Schwarzenegger. Does this terminate? Somebody.

STUDENT: [UNINTELLIGIBLE]

PROFESSOR ERIC GRIMSON: Ah, thank you, so it does terminate, right? You're sitting too far back, let me try-- oh, too far!-- Sorry. Come get me one later if you can't find it.

Yes, it stops at the first step, right? Let's look at it. It says, if answer, sorry, imagine x is -16, ANS is 0, is less than -16, no. So what does it do? It prints out 0.

Ah! So that now answers my second question, it does terminate, but does it give me the right answer? No. Right? It gives me an answer, and imagine I'm using this somewhere else, you know, it's going to go off and say, gee, the square root of -16 is 0. Well, it really should be a, you know, an imaginary number, but this is not a valuable thing to have come back.

So that's the second thing I've just highlighted here, is that I now have the ability to check whether it does the right thing.

And those are two things that you'd like to do with every looping construct you write: you'd like to be able to assure yourself that they will always terminate, and then the second thing you'd like to do, is to assure yourself that it does give you back a reasonable answer.

We started to talk about ways to do the former. It's looking at the end test. It's looking at the kinds of conditions you're going to put in. For the latter, this is a place where running test cases would do a good job of helping with that. Nonetheless, having done that, let's look at a better way to write this. Which is right here, it is also, I think, on your sheet, I'm going to uncomment that, and comment this one out, yeah. All right?

So let's look at this code for a second. Notice what this does. Certainly the heart of it, right in here, is still the same thing. But notice what this does. The first thing it does is, it says, let's check and make sure x is greater than or equal to 0. If it isn't, notice what's going to happen. None of that block is going to get executed, and it's going to come down here and print out a useful piece of information, which says, hey, you gave me a negative number. I don't know how to do this.

If it is, in fact, positive, then we're going to go in here, but now notice what we're doing here. There is the basic thing we did before, right? We're checking the end test and incrementing, actually I was going to, I commented that out for a reason you'll see in a second, but I, normally I would keep this on, which would let me, at each step, see what it's doing. If I ran this, it would print out each step. Which is helping me make sure that it's incrementing the right way.

OK, once it gets to the end of that, what's it going to do? It's going to come down here and, oh. What's that doing? Well, I cheated when I started. I said, somebody's giving me a perfect square, I'm looking for the square root of it. But suppose I gave this thing 15, and asked it to run. It'd still give me an answer. It just would not be the answer I'm looking for.

So now, in this case, this code is going to, when we get here, check, and if you haven't seen that strange thing there, that exclamation point in computer-ese called a bang, it says if ANS star ANS is not equal to x, all right? What's that say, it says, I've already gotten to the end of the loop, I'm now past where I wanted to be, and I'm going to check to make sure that, in fact, this really is a perfect square. If it isn't, print out something says, ah, you gave me something that wasn't a perfect square. And only if that is true, am I going to print out the answer.

It's the same computation. But this is a nice way of writing it, often called defensive programming. And I think we have lots of variations on it-- I don't know about John, what your favorite is, for the definition of defensive programming-- for me it says, make sure that I'm going through all possible paths through the code, make sure I'm

printing out, or returning if you like, useful information for each style, sorry, for each path through the code, make sure that for all possible inputs there is a path through the code, or a way to get through the code, that does not cause an error or infinite loop. What else would you add, John?

PROFESSOR JOHN GUTTAG: Well, we'll come back to this later in the term, and talk in some detail about particular techniques. The basic idea of defensive programming is, to assume that A, if you're getting inputs from a user, they won't necessarily give you the input you've asked for, so if you ask for a positive number, don't count on them giving you one, and B, if you're using a piece of a program written by a programmer who is not perfect, perhaps yourself, there could be mistakes in that program, and so you write your program under the assumption that, not only might the user make a mistake, other parts of your program might make a mistake, and you just put in lots of different tests under the assumption that you'd rather catch that something has gone wrong, then have it go wrong and not know it. And we'll talk later in the term about dozens of different tricks, but the main thing to keep in mind is the general principle that people are dumb. And will make mistakes. And therefore, you write your programs so that catastrophes don't occur when those mistakes are made.

PROFESSOR ERIC GRIMSON: Good. As John said, we're going to come back to it. But that's what, basically the goal here. And you saw me put my hands up when I said stupid programmer? I've certainly written code that has this problem, I've tried to use my own code that has this problem, and good to us, right, good hygiene, I'm going to use that word again here, of getting into the habit of writing defensive code up front, it's part of that collection of things that you ought to do, is a great thing to do.

I stress it in particular because, I know you're all going to get into this stage; you've got a problem set due in a couple of hours, you're still writing the code, you don't want to waste time, and I'm going to use quotes on "waste time", doing those extra things to do the defensive programming, you just want to get the darn thing done. It's a bad habit to get into, because when you come back to it, it may haunt you later on down the road. So really get into that notion of trying to be defensive as you program.

OK. The other thing I want to say here, is that this style of program we just wrote, is actually a very common one. And we're going to give it a nice little name, often referred to as exhaustive enumeration.

What does that mean? It says, I'm literally walking through all possible values of some parameter, some element of the computation, testing everything until I find the right answer. All right, so it's, you know, again, I can even write that down, essentially saying, try all reasonable values until you find the solution. And you might say, well, wait a minute, isn't that going to be really expensive? And the answer is, yeah, I guess, if you want to search, you know, all the pages on Google, one by one, yes, probably, it's going to take a while. But there are an awful lot of computations for which this is the right way to do it. You just want to exhaustively go through things.

And just to give you a sense of that, let me show you an example. I'm going to change this, all right? Nice big number. You know, computers are fast these days. I can make this even bigger, it's going to do it fairly quickly, so it really is quick to do

this. It doesn't mean that exhaustive enumeration is a bad idea, it is often the right idea to use.

So we've seen one example of this, this idea of walking through all the integers looking for the square root. Let's look at some other examples, in order to try and see other ways in which we could do it.

OK. In particular, let's go over to here, and let me show you a second example. And let me comment that out. Here's another problem that I'd like to solve. Suppose I want to find all the divisors of some integer, I want to figure out what all the divisors are that go evenly into it. Again, same kind of reasoning says, given some value x, I happened to pick a small one here, what's an easy way to do this? Well, let's just start at one. That's my variable I'm going to change and check. Does it divide evenly into x? If it does, print it out. Move on to the next one, print it out. So again, I can do the same kind of thing here, you can see that, in fact, let's just run it to make sure it does the right thing, OK? In fact, if I go back to the code, what did I decide to do here? I say, starting with an initialization of I, there's my first step, as equal to 1, I'm going to walk through a little loop where I check, as long-- first of all, as long as I is less than x, so there's my end test, I'm going to do something. And in this case, the something is, I'm going to look to see if I divides x evenly. So I'll remind you of that amp-- sorry, that percent sign there, that says if x divided by I has a remainder, because this gives me back the remainder, if that's equal to 0, print something out. And there's my nice increment. Simple little piece of code. Notice again, exactly the same form: I picked the thing I wanted to vary, I initialized it outside the loop, I have a test to see when I'm done, and then I've got a set of instructions I'm doing every time inside the loop. In this case, it's doing the check on the remainder and printing them out. And when I'm done with the whole thing, before I end the increment of the variable, you know, when I'm done, I'm just not returning anything out. OK. So now you've seen two simple examples. Let me generalize this. In this case, my incrementer was just adding 1 to an integer, it's a pretty straightforward thing to do. But you can imagine thinking about this a little differently. If I somehow had a collection, an ordered collection of all the integers, from 1 to 10, I could imagine doing the same thing, where now what I'm doing is, I'm starting with the first element of that collection, doing something, going to the next element, doing something, going to the next element, doing something, I'm just walking through the sequence of elements. Right? And I haven't said yet, how do I get that collection, but you could certainly conceptualize that, if I had that collection, that would be nice thing to do. That is a more common pattern. That is basically saying, given some collection of data, I want to have again a looping mechanism, where now my process is, walk through this, the collection, one element at a time. And for that, we have a particular construct, called a FOR loop. It's going to do exactly that for us. It's going to be more general than this, and we're going to come back to that, in fact, Professor Guttag's going to pick this up in a couple of lectures, but we can talk right now about the basic form. The form of a FOR loop says, FOR, and I'm going to put little angle braces in here again, to say, for some variable, like a name I want to get to it, in some collection, and then I have a block of code. And what it's saying semantically is, using that variable as my placeholder, have it walk through this collection, starting at the first thing, execute that code, then the next thing, execute that code, and so on. One of the advantages of this is, that I don't have to worry about explicitly updating my variable. That happens for me automatically. And that's very nice, because this allows me to be sure that my FOR loop is going to terminate. And because, as long as this collection is finite, this thing is just going to walk through. All right? So, if I show you, for example, I'm going to comment this one out in the

usual manner, and let's look at uncommenting that, there is the same piece of code. Now, I slung something by you, or snuck something by you, which is, I hadn't said how to generate the set of integers from 1 to 10. So, range is a built-in Python function. I'm going to come back to it in a second. For now, just think of it as saying, it gives you all the integers from 1 up to, but not including, x. OK. But now you can see the form. This now says, OK, let I start as the first thing in there, which is 1, and then do exactly as I did before, the same thing, but notice I don't need to say how to increment it. It's happening automatically for me. OK. In fact, if I run it, it does the same thing, which is what I would expect. OK. Now, the advantage of the FOR, as I said, is that it has, then, if you like, a cleaner way of reading it. I don't have to worry about, do I initialize it, did I forget to initialize it outside the loop, it happens automatically just by the syntax of it, right there, that's going to start with the first element. I don't have to worry about, did I remember to put the incrementer in, it's going to automatically walk it's way through there. Second advantage of the FOR is, that right now, we're thinking about it just as a sequence of integers. We could imagine it's just counting its way through. But we're going to see, very shortly, that in fact those collections could be arbitrary. We're going to have other ways of building them, but it could be a collection of all the primes. Hm. There's an interesting thing to do. It could be a collection of, ah, you know, I don't know, batting averages of somebody or other. It could be arbitrary collections that you've come up with in other ways. The FOR is, again, going to let you walk through that thing. So it does not have to be something that could be described procedurally, such as add 1 just to the previous element. It could be any arbitrary collection. And if I were to use that again, I'd just put it on your handout, I could go back and rewrite that thing that I had previously for finding the square roots of the perfect squares, just using the FOR loop. OK. What I want to do, though, is go on to-- or, sorry, go back to-- my divisor example. [UNINTELLIGIBLE PHRASE] OK. Try again. I've got a number, I want to find the divisors. Right now, what my code is doing is, it's printing them up for me, which is useful. But imagine I actually wanted to gather them together. I wanted to collect them, so I could do something with them. I might want to add them up. Might want to multiply them together. Might want to do, I don't know, something else with them, find common divisors, of things by looking at them. I need, in fact, a way to make explicit, what I can't do that with range, is I need a way to collect things together. And that's going to be the first of our more compound data structures, and we have exactly such a structure, and it's called a tuple. This is an ordered sequence of elements. Now, I'm going to actually add something to it that's going to make sense in a little while, or in a couple of lectures, which is, it is immutable. Meaning, I cannot change it, and we'll see why that's important later on. But for now, tuple is this ordered sequence of structures. OK. And how do I create them? Well, the representation is, following a square bracket, followed by a sequence of elements, separated by commas, followed by a closed square bracket. And that is literally what I said, it is an ordered sequence of elements, you can see where they are. OK? So, let me do a little example of this. If I go back over here, let's define-- er, can't type-- I can look at the value of test, it's an ordered sequence. I need to get elements out of it. So again, I have a way of doing that. In particular, I can ask for the zeroth element of test. OK, notice, I'm putting a square bracket around it, and it gives me-- I know this sounds confusing, but this is a long tradition, it gives me-- ah, yes.

STUDENT: [UNINTELLIGIBLE]

PROFESSOR ERIC GRIMSON: Sorry?

STUDENT: [UNINTELLIGIBLE]

PROFESSOR ERIC GRIMSON: I created a list here? Ah, thank you. I'm glad you guys are on top of it. You're saying I want that. Is that right, John? Yes? OK. Sorry. You're going to see why this was a mistake in a little while. I did not want to make a list, I wanted to create a tuple thank you for catching it. I want parens, not square brackets there. You'll also see in a little while why both of these things would work this way, but it's not what I wanted. OK? So I guess I should go back, and let me do this correctly this way. Again, I can look at test, and I guess test now if I want to get the element out-- angle bracket or square bracket? I still want square bracket, that's what I thought-- OK. Now I can go back to where I was, which is a strange piece of history, which is, we start counting at 0. So the-- I hate to say it this way, the first element of this tuple is at position 0, or index 0, OK?-- so I can get the zeroth one out, I can get, if I do 2, I get the third thing out, because it goes 0, 1, 2-- notice, however, if I do something that tries to go outside the length of the tuple it complains, which is right. Tuples? also have another nice structure, which is, I can go the other direction, which is, if I want to get the last element of that tuple I give it a negative index. So, imagine, you think of it as, is it starting right, just before the beginning of the thing, if I give it a it's going to take the first one, if I give it a 1, it's going to take the next one, but I can go the other direction, if I give it a -1, it picks up the last element of the tuple.

And again, I can go -2, go back. So this is what we would call selection. We can do things like foo of to get out the particular element. I can also pick up pieces of that tuple. Again I want to show you the format here. If I give it this strange expression, this is saying I want to get the piece of the tuple starting at index 1, it's going to be the second element, and going up to but not including index 3. And it gives me back that piece. Actually a copy of that piece of the tuple. This is called slicing. And then just to complete this. Two other nice things you can do with slices are you can get the beginning or the end of tuple. So, for example, if I say TEST and I don't give it a start but I give it an end, then it gives me all the elements up to that point. And I can obviously do the other direction which is I can say skip to index 2 and all the remaining pieces. This lets me slice out, if you like, the front part or back part or a middle part of the tuple as I go along.

What in the world does that have to do with my divisor example? Well, actually, before I do that let me in fact fill in a piece here. Which is remember I said range we could think of conceptually as a tuple -- or sorry as a sequence of these things. In fact it gives me back, now I hate this, it's actually a list it's not a tuple. But for now think of it as giving you back an explicit version of that representation of all those elements. You'll see why I'm going to make that distinction in a couple of lectures.

All right. What does this have to do with my divisor example? This says I can make tuples, but imagine now going back to my divisor example and I want to gather up the elements as I go along. I ought to be able to do that by in fact just adding the pieces in. And that's what I'm going to do over here. Which is, let me comment that out, let me uncomment that. And I guess I need the same thing here, right? I need parens not, thank you. You can tell I'm an old time list packer. I really do love these things. And is that right, John? OK, so my apologies that your handout is wrong. I did not think to check about the difference between these things.

Nonetheless, having done that, let's look at what I'm going to do. I now want to run a loop where I need to collect things together. I'm going to give a name to that. And

what you see there is I'm going to call divisors initially an empty tuple, something has nothing in it. Right here. And then I'm going to run through the same loop as before, going through this set of things, doing the check. Now what I'd like to do, every time I find a divisor I'd like to gather it together. So I'm going to create a tuple of one element, the value of i. And then, ah, cool. Here's that addition operation that's badly overloaded. This is why Professor Guttag likes and I don't. Because given that this is a tuple and that's a tuple, I can just add them together. That is concatenate them, if you like, one on the end of it. And if I keep doing that, when I'm done divisor will be a collection of things. So let me just run it. All right. This is what I get for trying to --

STUDENT There should be a comment after the i in parentheses.

PROFESSOR ERIC GRIMSON: Thank you. Right there. All right, we'll try this again. OK. And there are the set of devices. Thank you. Who did that? Somebody gets, no? Yours? Thank you. Nice catch too by the way. All right, so now that you can see that I can screw up programming, which I just did. But we fixed it on the fly. Thank you. What have we done? We've now got a way of collecting things together, right? And this is the first version of something we'd like to use. Now that I've gotten that bound as a name, I could go in and do things with that. I could go in and say give me the fourth divisor, give me the second through fifth divisor. Again as I suggested if I've got two integers and I want to find common divisors I could take those two lists and walk through them. I shouldn't say list, those two tuples, and walk through them to find the pieces that match up.

So I've got a way now of gathering data together. The last thing I want to do is to say all right, now that we've got this idea of being able to collect things into collections, we've got the ability now to use looping structures as we did before but we can walk down then doing things to them, where else might we have this need to do things with looping structures? And I'm going to suggest you've already seen it. What's a string? Well at some level it is an ordered sequence of characters. Right? Now it is not represented this same way. You don't see strings inside these open parens and closed parens. You don't see strings with commas between them, but it has the same kind of property. It is in ordered sequence of characters. We'd like to do the same thing with strings. That is we'd like to be able to get pieces of them out. We'd like to be able add them together or concatenate them together. We'd like to be able to slice them. And in fact we can.

So strings also support things like selection, slicing, and a set of other parameters, other properties. And let's just look at that. Again if I go back here, let me comment this out. Right here are a pair of strings that I've set up, s 1 and s 2. Let me just run these. We can go back over here. So I can see the value of s 1, it's a string. I can do things like s 1 and s 2. As we saw before, it simply concatenates them together and gives me back a longer string. But I can also ask for parts of this. So I can, for example, say give me the first element of string 1, s 1. Ah, that's exactly what we would have thought if this was represented as an ordered sequence of things. Again I should have said first, index 0, the first one. I can similarly go in and say I'd like all the things between index 2 and index 4. And again, remember what that does. Index 2 says start a 0. 1, 2. So a, b, c. And then it goes up to but not including index 4 so it gets c and d and then it stops. I can similarly, just as I did with the tuples, I can ask for everything up to some point or I can ask for everything starting at some point and carrying on.

Now what you're seeing here then is the beginning of complex data structures. And the nice thing is that there's a shared behavior there. Just as I can have tuples as an ordered collection of things, strings behave as an ordered collection of things. So I can start thinking about doing manipulation on strings. I can concatenate them together, I can find pieces inside of them, I could actually do things with them. And let me show you just a simple little example of something I might want to do. Suppose I take, I better comment this one out or it's going to spit it out. Let me comment that out. Suppose I take a number. I'd like to add up all the digits inside of the number. I can use the tools I've just described in order to capture that.

So what would I want to do? I'd like to somehow walk down each of the digits one at a time and add them up. Ah, that's a looping mechanism, right? I need to have some way of walking through them. An easy way to do it would be inside of a FOR. And what would I like to do? Well I need to take that number and I'm going to turn it into a string. So notice what I'm going to do right here. I take that number and convert it into a string. That's an example of that type conversion we did earlier on. By doing that it makes it possible for me to treat it as an ordered sequence of characters. And so what's the loop going to do? It's going to say FOR c, which was my name for the character in that string. That means starting at the first one, I'm going to do something to it. And what am I'm going to do? I'm going to take that character, convert it back into an integer, and add it into some digits. And I've done a little short hand here, which is I should have said some digits is equal to some digits plus this. But that little short hand there is doing exactly the same thing. It is adding that value into some digits and putting it back or signing it back into some digits. And I'll walk through that loop and when I'm done I can print out the total thing does. And if I do that, I get out what I would expect.

So what have I done? We've now generalized the idea of iteration into this little pattern. Again as I said this is my version of it, but you can see, every one of the examples we've used so far has that pattern to it. Figure out what I'm trying to walk through. What's the collection of things I'm trying to walk through. Figure out what I want to do at each stage. Figure out what the end test is. Figure out what I'm going to do at the end of it. I can write it explicitly. I can write it inside of a FOR loop. And we've started to add, and we'll see a lot more of this, examples of collections of structures so that we don't just have to do something that can be easily described as walking through a set of things but can actually be a collection that you walk through.

The last thing I want to point out to you is, I started out with this list. I haven't added anything to the list, right? I mean I've got a different kind of looping mechanism. I guess I should say that's not quite true. I've added the ability to have more complex data structures here. But I dropped a hint in the first lecture about what you could computer with things. In fact if you think for a second about that list, you could ask what can I compute with just that set of constructs? And the answer is basically anything. This is an example of what is referred to frequently as being a Turing complete language. That is to say with just those set of constructs, anything you can describe algorithmically you can compute with that set of constructs. So there's good news and bad news. The good news is it sounds like we're done. Class is cancelled until final exam because this is all you need to know, right? The bad news is of course that's not true. The real issue is to figure out how to build constructs out of this that tackle particular problems, but the fundamental basics of computation are just captured in that set of mechanisms. All right, we'll see you next time.