MIT OpenCourseWare
http://ocw.mit.edu

6.00 Introduction to Computer Science and Programming, Fall 2008
Transcript – Lecture 4

PROFESSOR ERIC GRIMSON: As I've done in the previous lectures, let me set the stage for what we've been doing, so we can use that to talk about what we're going to do today.

So far, we have the following in our language. Right, we have assignment. We have conditionals. We have input/output. And we have looping constructs. These are things like FOR and WHILE loops. And of course we've got some data to go with that.

One of the things we said last time was, with that set of things, the ability to give values-- sorry, to give names to values-- the ability to make decisions, the ability to loop as a function of that, the ability get things in and out, we said that that actually gave us a language we said was Turing-complete. And that meant, in English, that this was enough to write any program.

Now that's a slight lie-- or actually in these days of political debates, a slight misspeaking, a wonderful word-- it is technically correct. It is enough to allow us to write any program, but it's not enough to allow us to easily write any program.

And so I joked, badly, I'll agree, at the end of last lecture, that we can just stop now, go straight to the final exam, because this is all you need to know.

The point is, yes it's enough to start with, but we want to add things to this that let us problem solve well. And one of the things to think about is, even though I've got all of that, let's think about what I could do, if I wanted to write a piece of code.

Right now, you've got to write it in one file. It's a long sequence of instructions, it starts at the beginning, walks through, may jump around a little bit, but eventually comes down at the end. It's okay for the things you're doing for the early problem sets. Ten lines of code. Twenty lines of code.

Imagine instead, you're writing code that's a hundred thousand lines, a million lines, of code. You don't want to write it in this form. All right, and the reason you don't want to do that is, well, several.

First of all, it's really hard to figure out where everything is, where everything goes, making sure I'm in the right place. To use an ancient expression from the 1970's, which only John and I will appreciate, it's really hard to grok what that code is doing, to understand what it's trying to make happen.

And the reason that that's the case is, what we don't have, are two important things. We don't have decomposition, and we don't have abstraction. And that's what we're going to add today.

So what does that mean? Those are fancy terms. Decomposition is a way of putting structure onto the code. It's a way of breaking the code up into modules. Modules that makes sense on their own. Modules that we can reuse in multiple places. Modules that, if you like, isolate components of the process.

And abstraction is related to that, abstraction is going to let us suppress details. It's going to let us bury away the specifics of something, and treat that computation like a black box. And by black box, I mean, literally behaves like a mysterious little black box. You put some inputs in, it has a contract that says if you put the right kind of inputs in you'll get a specific output coming out, but you don't have to know what's inside of that box. And that abstraction is really important.

Again, imagine if I'm a writing a piece of code. I want to just use it, I shouldn't have to worry about what variables I use inside of it, I have shouldn't have to worry about where that is in the code, I should be able to just abstract it away. And that's what we want to add today, are those two things.

Now, our mechanism for doing that-- or at least one mechanism, I shouldn't say the only one-- one mechanism for doing that is going to be to add functions to our language. Now, the point of a function is that it's going to provide both of these things, so the first thing it's going to do is, it's going to let us break up into modules.

Second thing they're going to do is let us suppress detail. And in essence what that does is, the functions, and we're going to look at a bunch of examples in a second, these functions are going to give us a way to, in some or in one way of thinking about it is to create new primitives. And I'm going to put those in quotes, it's a generalization.

What do I mean by that? The idea of a function is, that I'm going to capture a common pattern of computation. Computing square root. I'm going to capture it in a piece of code, I'm going to be able to refer to it by a name, and I'm going to suppress the details, meaning inside of that computation, you don't need to know what it does. You just need to know, if I give it the right kind of input, it'll give me back an input that satisfies the contract that I set up.

And that in essence says, I've just created the equivalent of a new primitive. Same way that I have multiplication or division as a primitive, functions are going to give me, or somebody else who wrote them for me as part of a library, a new primitive that I'm going to be able to use. And that gives me a lot of power in terms of what I want to have inside of the language. OK.

So, let's look at an example. To try to see what we're going to do with this. Before I do that though, let me try and give you an analogy to keep this in mind of why we want to basically build these abstractions and what we need in order to have them work together.

So here's the supposed to say silly analogy. You can tell my jokes are always bad. John's are much better, by the way, which is why Thursday will be a much better lay-- a better lecture.

But here's the example. You've been hired by PBS to produce a nice thirteen-hour documentary, or drama, that's going to run. And, you know, you start by saying, OK, thirteen hours, I'm going to break it up into thirteen different chunks. I'm going to assign each chunk to a different writer. And they're going to go off and write that element, that hour's worth of stuff. You can imagine what you get: each hours worth of drama, if you like, may be great, but it may have absolutely nothing to do with the other twelve hours. And unless, you know, you've been hired to do Pirandello's Six Characters In Search Of An Author, this is not a great thing, because you get something that is really confusing.

Now, what's the point of the analogy? What do I need for those writers to all interact together? I need a specification. I need a contract that says, here's what I want in terms of things that you're going to take as input, to begin your part of the drama, here's what you're going to produce at the output, and the details of what they do inside are up to them.

An idea of abstraction, that idea of specification, is exactly what we want to use inside of our functions. We won't make you write dramas like Pirandello, but we're going to try make you at least write good code. And that's we're going to try and do.

All right. Let's set the stage for it. Up on the screen, I've got-- I commented it out, but I've got a piece of code that you've seen before, right up here. OK? What is that? It's the piece of code we wrote for computing square roots, square roots of actually perfect squares. [UNINTELLIGIBLE]

Just to remind you what it does, we bound x to some value, we set up an initial variable called ANS or answer, and then we run through a little loop. All right, we're-- well actually, I should say that better, we first check to see, is x greater than or equal to zero, if it's not, then we come down here and we print something out, otherwise we run through a little loop to get the answer, and then we check it and we spit something out. It does the computation, that's fine.

Suppose I want to compute square roots a lot of places in a big chunk of code. Right now, I have to take that piece of code and replicate it everywhere I want in my larger file. And I've got to worry about, is somebody else using ANS, answer, as a variable, in which case I've got to be really careful. Is somebody else using x as a variable? I've got to deal with a lot of those details.

I want to abstract that. And the abstraction you see, right here. I'm going to highlight it for a second so you can see it. I want you to look at it on the handout as well. This is the creation of a function.

And I want to describe both the syntax, what we're doing, and then the semantics of how do we use it and what does that mean. So.

Here's the syntax of the function. First of all, we have a keyword. def. Definition or define, depending on which sort of piece of history you come from. This is a keyword to Python that says, when it reads this in the file, it says, I'm creating a definition. I'm creating a function.

And that's follow-- so this is, let me say this is a keyboard-- that is followed immediately by a name. And this equates to that. In this case, sqrt, square root. I'm

saying, this is the name I'm going to give to this function. This is the name to which I'm going to refer when I want to use this function. All right?

And notice, immediately after that name, we have an open and close paren with another variable name inside of that. And this defines formal parameters of this function. Yup.

PROFESSOR JOHN GUTTAG: [INAUDIBLE]

PROFESSOR ERIC GRIMSON: It does indeed, thank you. This is me being a Scheme hacker, not a Python hacker. Yes, def has to be lowercase or won't recognize it. Thank you, John. OK. def's the keyword. I'm creating a function. sqrt-- again, I'm being careful about case-sensitive, I'm using all lowercase here, followed by an open paren, and I said, formal parameters. We'll see there could be more than one there. We're going to come back to what they mean in a second, but for now, think of them as, or think of x, in this case, as the place holder. This place holder is saying, if you give me a value for x, inside the body of this function I'm going to use that value everywhere I see x. Question.

STUDENT: [INAUDIBLE]

PROFESSOR ERIC GRIMSON: Ah, we're going to come back to this in a second. But the question was, do I always need an input? I can have functions with no parameters, that's fine, I will still need the open and close paren there to identify that I have no parameters. We're going to see an example in a second.

Good question. Actually, I've got to get rid of this candy, so since it was a good question, here you go. Nice catch. Almost. Sorry. OK. No, I'm not. I'm sorry. I thought you had it, and then I've got the wrong glasses on and I realized you didn't, so I will, ah come back to that later.

What are we doing here? We got definition, we got name, we got a set of formal parameters. Right. If you look at the rest of that code, gee, it looks a lot like what I had elsewhere. Of what I had outside of it, right? It's running through a similar set of loops. So in some sets, as long as x has the value I want, it ought to do the right thing.

However, there's a couple of other changes there that we want to highlight. In particular, notice-- let me highlight it for you, if I can find it with the wrong glasses on-- we've got these return commands. So return is another keyword. And it basically says, when you get to this point in the computation, stop the computation. Literally, return the control from this function, and take the value of the next expression, and return that as the value of the whole computation.

Now, the one that we're most interested in is the one where, in fact, it gets out ANS, so you see down here in the code, there's a spot where it's going to return the value of ANS, which is what we want, right? That's the thing that holds the value that we intended to have.

But there's another couple of places in that code where it's got this funky-looking thing, return none, and notice none's in a different color. None is a special value, and it has the following slightly-odd behavior: it is a value, we can return it-- God bless--

but what none says is, there is no value coming back from this computation. So when it is returned, and we'll see this in a second, to the interpreter, it doesn't print.

OK. It simply doesn't print anything. Nonetheless, it is actually a value and we can use it, for example, to do comparisons. If we want to know, did this function return a value or not, rather than reserving, say, -1 or some other special thing which you might want to use some other ways, it literally returns this very special value that says, there is no actual value return from this computation.

OK. Note, by the way, if I chase through each possible path, like there's some IFs in here, there's some places to go, at least in this piece of code, every possible path through this code ends in a return. And that's a good programming discipline, to make sure that happens.

There's an exception, which we'll see in a second, but I'll highlight, which is, if we get to the end of the procedure, there's sort of an implicit return there. In fact, a return of none, in that case. It comes out of it.

But if I look at this, right? If I come into this code, I'm going to check this branch first, if it's not true, ah, there's a return at the end of that branch. If it is true, I do that, and then I've got a second test. If it's true, I return, otherwise a return. So there's a return branch on every possible path through the code. And that's valuable, it's something you want to think about as your right your own.

OK. What do I do to use this, in particular? How do I invoke this? OK, so I'm going to invoke a function by passing in values for the parameters. And in this case, that literally means typing sqrt, with some value inside the parens.

OK. Now, let's just try this out to see what happens. I'm going to make sure I've got it there, so if I type, for example, sqrt of 16, ah-ha! What did it do? Well, let's talk about what it did. What this invocation does, is the following: it binds, and I'm going to say this specifically for this example, rather than general, it binds x to 16. Just as you would have done with an assignment statement up in the top level thing.

But this binding is local. Meaning it only holds within the confines of the code of this procedure. Relative to that, think of that as creating what we, I'm going to call a new environment. Relative to that, it does all the other execution we would do, including, notice the first instruction there is to the set up a binding for ANS.

So answer, or ANS, is also bound only locally. Meaning, inside the confines of this environment of this procedure. ANS starts off with a value of and now we just run through that loop just like we did before. Writing Increments it slowly, checking to see if ANS squared is bigger than x, and when it gets to that point, it checks to see, is it actually a perfect square or not, and it returns it. And once it returns it, it returns a value from that return, that in this case is just printed out.

All right. Now I want to say a couple of things about these local bindings. I'm going to repeat this a second time, because it's important. These local bindings do not affect any global bindings.

What does that mean? Let me show you a little example, and then we'll come back to this. I've got a little function here. See, I've defined f of x to be a function that

takes a value of x in, changes x to x+1, and then just returns the value. OK. So it's just adding 1 to x.

But I want you to see now what happens if I use this. Let's bind x to the value of 3. It's creating a binding for x in this global environment. This is what the interpreter sees. All right? In fact, if I look at x, its value is 3. Let's bind z eh let's bind z to the-- if I could type it would help-- say, f of 3. OK? So the value is z is 4, it's what I expect, right? Locally x got bound to 3, I added 1 to it, whoop-dee-doo, I get back a 4. But what's the value of x? It's still 3.

The way to think of this is, again, I've got multiple scopes, or multiple frames, or if we're going to come back to those terms, I'm going to use the word environment, because I'm an old-time Lisp hacker, multiple environments in which there are bindings.

So let me spell this out in just a little bit more detail. What this is saying is the following. When I'm talking to the interpreter, when I'm typing things in as I just did, to that Python environment, I'm getting what I'm going to call global bindings.

I'm going to draw a little chart here. Think of this as the, as the world of the interpreter, in that I've got things like x bound to the value of 3.

When I call or invoke a function, think of it as creating a local table. Inside that local table, I bind the formal parameter, which is what I do I did 16 right to some value. This x only gets seen by sqrt. Inside of there, I can bind other things, like ANS gets locally bound to 0, and then it increments around and eventually we return that value out. When I get to a return from sqrt, some value is returned back to the interpreter, and that table goes away. But that table does not affect any bindings for other instances of the variable like x for ANS.

OK. Let's look at a couple of examples, just to sort of stress that. And one of the things I wanted to show is, OK. Again, I can now use a function just as if it was a primitive, so this is just an assignment and I going to take test to be the value of that, of course nothing gets printed because that was an assignment statement. All right? So if I called sqrt alone, that return value is done, but in this case I bound it to test, so I can go look at test, and there it is.

What happens if I do that? OK. If you look at the code, it printed out, it's not a perfect square, which is what I wanted, but now, what's the value of test? OK, I bound test to something, if I look at it, it doesn't print anything, but-- if I could type-- I can ask, is test bound to that special name none? The answer is yes.

Boy, this seems like a nuance, right? But it's a valuable thing. It says, in each case, I return some useful value from this procedure. I can check it, so if this was part of some other computation, I want to know, did it find a perfect square or not? I don't have to go read what it printed out in the screen. This has returned a value that I can use. Because I could do a test to say, is this a return value? If it's not, I'll do something else with it. So the binding is still there, it simply doesn't print it out.

OK. What do we have out of this? Simple, seems like, at least addition. We've added this notion of a function.

I've highlighted some of the key things we got here, right? We have that def keyword, we've got a name, we've got a list-- or I shouldn't say a word list, we have a collection of formal parameters that we're going to use-- we have a body, and the body looks just like the normal instructions we'd use, although by the way, we ought to be able to use functions inside the body, which we're going to do in a second, and then we're going to simply return some values out of this.

Now I started by saying, build these functions. I'm trying to get both decomposition and abstraction. Well, you hopefully can see the decomposition, right? I now have a module.

OK, let me set the stage. Imagine I wanted to do sqrt, or square root-- no, I'm going to use sqrt, that's the name I'm using here-- square root a hundred different places in some piece of code. Without function, I'd have to copy that piece of code everywhere. Now I got one just simple thing, and I simply have isolated that module inside of that function.

What about abstraction? Well, I've got part of what I want for abstraction. Abstraction, again, says I'm going to suppress details.

Now that I've written sqrt, I can just use it anywhere I want in the code. You've got to rely on the fact that I wrote it correctly, but you can basically suppress the details of how it's used.

There's one more piece that we'd like to get out of that, and that is-- you may have been wondering, what's with the funky stuttering here of three double-quotes in a row. All right? And that is a specification. Which is a really valuable thing to have.

So what is the specification going to do? It is my place, as a programmer, to write information to the user. This is me writing one hour of that episode of Pirandello and telling the other authors, here's what I'm assuming as you use it. So it's up to me to do it right, but if I do it, I'm going to specify, what does this function do? What does it expect as input, and any other information I want to pass on.

And notice, by the way, if I do that, I'm going to come down here, and I type sqrt and open the paren, ah-ha! It shows me what the creator, in this case actually I stole this from John so what Professor Guttag put up as his specification for this piece of code.

Now, it's not guaranteed it's right, right? You're trusting the programmer did it right, but this now tells you something.

What is this? This is a wonderful piece of abstraction. It is saying, you don't need to know squat about what's inside the body of this function. You don't have to worry about the parameter names, because they're going to be preserved, you don't need to worry about how I'm doing it, this tells you how you can use this, in order to use it correctly. Of course, I can then close it off, and off we go.

All right, so that notion of abstraction and I was going to come back-- we're going to come back to multiple times during the term-- and it's not just abstraction, it's the idea of a specification.

And just to look ahead a little bit, you could easily imagine that I might want to not just put a statement in there, what the specs are, I might want to put some constraints. Some specific things to check for, to make sure that you're calling the code right. And it becomes a powerful way of reasoning about the code, a powerful way of using the code, so those notions of specs are really important.

Look, part of the reason I'm flaming at you is, something like square root, it seems dumb to write specs on it. Everybody knows what this is going to do. But you want to get into that discipline of good hygiene, good style. You want to write the specs so that everybody does in fact know what this piece of code is doing, and you're writing it each time around.

OK. Now that we've got functions, let's see what we can do as a problem-solving tool using them. In a particular, I've already said I want to get this notion of modularity, it's a module I can isolate, and I want to get the notion of abstracting away the details, let's see how we can actually use that to actually write some reasonably interesting pieces of code, but in particular, to see how we can use it to capture the ideas of decomposition and abstraction.

So I'm going to shift gears. Start with a simple problem. Boy, we're suddenly be transported to Nebraska. Or where I grew up, Saskatchewan. All right, we've got a farm air problem. I got a farmer, walks out into his yard, one morning. This farmer has a bunch of pigs in a punch-- it's been a long day-- a bunch of pigs and a bunch of chickens. And he walks out into the farmyard and he observes 20 heads and 56 legs. And for sake of argument, there are no amputees among the chickens and the pigs.

And the question is, so how many pigs does he have, and how many chickens does he have? Wow. What a deep problem, right? But you're going to see why we're going to use this in a second. So you know how to solve this, this is a fifth-grade problem, right?

And what's the way to solve this? System of linear equations. What are the equations here? Well, I could say, you know, the number of pigs plus the number of chickens equals 20, right? Because we've got 20 heads. And then what else do I have? Four times the number of pigs plus two times the number of chickens, assuming they're not next to a nuclear reactor, is 56.

And then how would you solve this? Well, it's, you sort of know how you'd do it if this was grammar school right? You'd pull out your pencil and paper, you can do it as a matrix inversion if you know how to do that, or you can just simply do substitution of one equation into another to solve it.

That's certainly one way to do it, but for computers that's not necessarily the easiest way. So another way of solving it is to do something we already saw last time, which is basically, why not simply enumerate all possible examples and check them? You could say, I could have zero chickens and 20 pigs, does that work? I've got one chicken and nineteen pigs, does that work? I've got two chickens and eighteen pigs, you get the idea.

So I'm going to solve this by enumerate and check, which is an example of what's called a brute-force algorithm. Meaning, I'm just going to write a little loop that does that. All right, so let's go back to our code. That's right, let me pull this over a little

bit, so I can see it. And what I'd like you to look at, I'm going to highlight it just for a second here, is those two pieces of code. OK?

Let's start with solve. OK. Here's the idea of solve. I'm going to have it take in as input how many legs I got, how many heads do I have, and I just want to write a little loop.

OK. I know how to do that, right? Write a little loop, all I'm going to do, is run a FOR loop here. I'm going to let the number of chickens be in this range. Remember what range does, it gives me a set or a collection or a tuple of integers from up to 1 - is the last value, so it's going to give me everything from up to the total number of heads.

Knowing that, I'm going to say, OK, how many pigs are there, well that's just how we're, however many I had total, minus that amount, and then I can see, how many legs does that give, and then I can check, that the number of legs that I would get for that solution, is it even equal to the number of legs I started with, ah! Interesting. A return.

In particular, I'm going to return a tuple. So, a pair or collection of those two values. If it isn't, then I'm going to go back around the loop, and notice what happens. If I get all the way around the loop, that is, all the way through that FOR loop and I never find a path that takes me through here, then the last thing I'm going to do is return a pair or a tuple with a special simple number none twice.

Yep. Are you telling me I want parens there and not, and not braces? All right. I hate this language, because I always want to have parens. Every time you see a square bracket, put a paren in. All right? Thank you, Christy. I'll get it eventually .

Having done that, right, notice what I've got. First of all, two parameters. It's OK. All it says is, when I call this, I need to pass in two parameters for this to work. All right? Now, if I want to use that, I'm going to use a second piece of code here, called Barnyard. I'm going to read in a couple of values, convert them into integers, and then I'm going to use solve to get a solution out.

And what do I know about solve? It is going to give me back a tuple a collection of two things, and so check out the syntax. I can give two names, which will get bound to the two parts of that return tuple.

OK, pigs will be the first part, chickens will be the second part. OK, and then once I've got that, well, notice: I can then check to see, did I return that special symbol none? Is the first part. That says, I took the branch through here that eventually got to the end and said, there wasn't a solution, in which case I'm

going to print out, there ain't no solution, otherwise I'll print out the pieces. All right, let's check it out. Ah, what did I say? Twenty and 56, Right? OK, notice the form. I've got two parameters, they're separated by a comma. Ah, right. Sorry? Yeah, but I see-- it's legs and heads, but it should not still have--

Oh, sorry. Thank you. I've been doing the wrong thing. I want Barnyard this way, and if I had looked when I opened the paren, it would have shown me a closed paren with no parameters. Aren't you glad I make mistakes, so you can see how well I can fix from these? All right.

Now I call that, and it says, tell me how many heads you want, give it a 20, and tell it how many legs you want, give it 56, and it prints out the answers. I know, whoop-dee-doo. But notice what's inside if here.

First of all, notice the modularity. I've used solve. All right? Right there. I've captured it as a computation. It's buried away, all the details are suppressed. I can use that to return values, which I can then use elsewhere, which I did-- and if I just come back and highlight this-- inside of that computation.

But I don't have to know, inside of Barnyard, what the values are used inside of solve. I don't know what the names of the variables are, I don't care, I can basically suppress away that detail.

Second thing we saw is, that using this as a computation, I can return multiple values. Which is actually of real value to me here as I use that. OK. Yeah. Question.

STUDENT: [INAUDIBLE]

PROFESSOR ERIC GRIMSON: Ah. The question was, when it returns, how does it distinguish between local and other things? So let me try and answer that.

Inside of solve, solve creates an environment where inside of that, it has bindings for the parameters it's going to use. All right? Like, number of-- wait, what did we call a were solve-- number of legs and number of heads. OK, those are bound locally. When solve is done, it wraps up, if you like, a value that it returns. Which is that.

That expression, or that value, or that value, literally gets passed back out of that local environment to the value that comes back out of it. So in particular, what's solved returns is a pair. It could be the pair of none, none, it could be the pair of, you know, whatever the answer was that we put up there. That value comes back out and is now available inside the scope of Barnyard. OK. And Barnyard then uses that. Question?

STUDENT: [INAUDIBLE]

PROFESSOR ERIC GRIMSON: Here? So the question is, why is this return on the same level as the FOR? Why do you think?

STUDENT: [INAUDIBLE]

PROFESSOR ERIC GRIMSON: No. Good question. All right?

So what's going to happen here? If I'm inside this FOR, OK, and I'm running around, if I ever hit a place where this test is true, I'm going to execute that return, that return returns from the entire procedure. OK? So the return comes back from the procedure.

So the question was, why is this return down at this level, it says, well if I ever execute out of this FOR loop, I get to the end of the FOR loop without hitting that branch that took me through the return, then and only then do I want to actually say, gee, I got to this place, there isn't any value to return, I'm going to return none and none.

I'm still trying to get rid of this candy, Halloween's coming, where were we? There's one, thank you. I don't think I'm going to make it, I did. Thank you.

Make sense? The answer is no, I want parens to create tuple and I get really confused about the difference between lists and tuples. For now, the code is working. Yes is the answer, all right? And we're having a difference of opinion as to whether we should use a tuple or a list here, right?

But the answer is yes, you can. And my real answer is, go try it out, because obviously you can tell I frequently do this the wrong way and the TAs give me a hard time every time. John.

PROFESSOR JOHN GUTTAG: Is the microphone on?

PROFESSOR ERIC GRIMSON: Yes.

PROFESSOR JOHN GUTTAG: As you'll see next week, tuples and lists are very close to the same thing. In almost any place where you can get away with using tuples you can use lists.

PROFESSOR ERIC GRIMSON: Yes.

PROFESSOR JOHN GUTTAG: But want to emphasize word is almost, because we'll see a couple of places where if it expects a tuple and you use a list you'll get an error message. But we'll see all that next week.

PROFESSOR ERIC GRIMSON: Right, when the real pro comes in to pick up the pieces I'm leaving behind for him. OK. Let me pull this back up. What we're doing now is we're building this encapsulation. Now one of the things you notice here by the way is, you know, this in essence just solves the simple problems. Suppose I now add one other piece to this. The farmer is not keeping a great set of things so in addition to pigs, and chickens he raises spiders. I have no idea why. He's making silk I guess. Right? Why am I giving you this example? I want to show you how easy it is to change the code. But, notice, once I've added this I actually have a problem. This is now an under-constrained problem. I have more unknowns than I have equations. So you know from algebra I can't actually solve this. There may be multiple solutions to this. What would I have to do to change my code? And the answer is fortunately not a lot.

So I'm going to ask you to look now at this set of things, which is solve 1 and Barnyard 1. OK. The change is, well, on Barnyard 1 it looks much the same as it did for Barnyard. Right, I'm going to read in the values of the number of heads and the number of legs. I'm going to use solve 1 as before, but now I'm going to bind out three variables. And then I'm going to do a similar thing to print things out. But would the solver do? Well here what I'd like to do is to run through a couple of loops. Right, how would I solve this problem? You can use the same enumerate and check idea, but now say gee let me pick how many pigs there are. Is that the one I used first? Sorry, let me pick the number of spiders there are. Having chosen the number of spiders, let me pick how many chickens I have. With those two in place, I now know how many pigs I must have and I can run through the same solution. The reason I'm showing you this is this is another very standard structure. I now have two nested loops. One running through a choice for one parameter, another one

running through a choice for a second parameter. And then the rest of the solution looks much like before. I'm going to get the total number of legs out. I'm going to check to see if it's right or not. And again I'm going to return either a three tuple there or a three tuple there. It's part of what I want, because I'm going to bind those values out.

And if I run that example, Barnyard 1, I don't know we'll give it 20 heads, 56 legs; and it find a solution. I ought to be able to run something else. I don't know, give me some numbers. How many heads? Pick an integer, somebody.

STUDENT: 5.

PROFESSOR ERIC GRIMSON: 5. All right. How many legs? 10? All right. We got an easy one. Let's just for the heck of it -- I should have found some better examples before I tried this. No mutant spiders here. OK, so what have I done? I just added a little bit more now. I'm now running through a pair of loops. Again notice the encapsulation, that nice abstraction going on, which is what I want. Once I get to this stage though by the way, there might be more than one solution. Because in an under-constrained problem there could be multiple solutions. So suppose I want to capture all of them or print all of them out.

Well I ought to be able to do that by simply generalizing the loop. And that's what the next piece of code on your a hand out shows you. I'm just going to let you look at this. If you look at solve 2, it's going to run through the same kind of loop, printing out all of the answers. But it's going to keep going. In other words it doesn't just return when it finds one, it's going to run through all of them. All right? Sounds like a reasonable thing to do. Notice one last piece. If I'm going to do that, run through all possible answers, I still want to know, gee, what if there aren't any answers? How do I return that case? And that shows you one other nice little thing we want to do, which is if I look in this code notice I set up a variable up here called Solution Found, initially bound to false. The rest of that code's a pair of loops. Pick the number of spiders. Pick the number of chickens. That sets up the number of pigs. Figure out the legs. See if it's right. If it is right, I'm going to print out the information but I'm also going to change that variable to true. And that allows me then, at the end of that pair of loops when I get down to this point right here, I can check to see did I find any solution and if not in that case print out there is no solution. So this gives you another nice piece which is I can now look for first solution, I can look for all solutions, and I can maintain some internal variables that let me know what I found. A trick that you're going to use a lot as you write your own functions.

All right, I want to end up with the last 10 minutes with a different variation on how to use functions to think about problems. And that is to introduce the idea of recursion. How many of you have heard the term used before? How may have you heard the term used before in terms of programming languages? Great. For the rest you, don't sweat it. This is a highfalutin term that computer scientists use to try and make them look like they're smarter than they really are. But it is a very handy way of thinking about, not just how to program, but how to break problems down into nice sized chunks. And the idea behind recursion I'm going to describe with a simple example. And then I'm going to show you how we can actually use it. The idea of recursion is that I'm going to take a problem and break it down into a simpler version of the same problem plus some steps that I can execute. I'm go to show you an example of a procedure, sorry a function, in a second. But let me give you

actually an analogy. If you look at US law, and you look at the definition of the US legal code that defines the notion of a natural born US citizen. It's actually a wonderful recursive definition. So what's the definition?

If you're born in the United States you are by definition a natural born US citizen. We call that a base case. It's basically the simplest possible solution to the problem. Now if you were not born in the United States, you may still be, under definition, a natural born US citizen if you're born outside this United States, both of your parents are citizens of the United States and at least one parent has lived in the United States. There's a wonderful legal expression. But notice what that is. It's a recursive definition. How do you know that your parents, at least one of your parents satisfies the definition? Well I've reduced the problem from am I a natural born US citizen to is one of my parents a natural born US citizen? And that may generalize again and it keeps going until you either get back to Adam and Eve, I guess. I don't think they were born in the US as far as I know, or you find somebody who satisfies that definition or you find that none of your parents actually are in that category.

But that second one is called the inductive step, or the recursive step. And in my words it says break the problem into a simpler version of the same problem and some other steps. And I think this is best illustrated by giving you a simple little piece of code. I use simple advisedly here. This is actually a piece of code that is really easy to think about recursively and is much more difficult to think about in other ways. And the piece of code is suppose I have a spring and I want to know if it's a palindrome. Does it read the same thing left to right as right to left. OK? How would I solve that? If the string has no elements in it it is obviously a palindrome. If the string has one element in it, it's a palindrome. There's the base case. If it's longer than one, what do I want to do? Well I'd like to check the two end points to see are they the same character? And if they are, then oh, I just need to know is everything else in the middle a palindrome?

I know it sounds simple, but notice what I just did. I just used a recursive definition. I just reduced it to a smaller version of the same problem. That is if I can write code that would solve all instances of smaller size strings, then what I just described will solve the larger size one. And in fact that's exactly what I have. I would like you to look at this piece of code right here called isPalindrome. Notice what it says. I'm going to pass in a string, call it s, binds it locally, and it says the following. It says if this is a string of length or 1, I'm done. I'm going to return the answer true. Otherwise I'm going to check to see is the first and last, there's that - 1 indexing, is the first and last element of the string the same? And if that's true is everything in the string, starting at the first element and removing the last element, a palindrome? Let me remind you. By saying first element remember we start at as the initial indexing point. Wonderful recursive definition.

OK, let's try it out. Go back over here and we're going to say isPalindrome. How did I actually spell this? Palindrome with a capital P. Only in New York, in Canada we pronounce it Palindrome. When you're teaching it you get to call it your way, I'm going to call it my way. Sorry John, you're absolutely right. OK. Notice by the way, there's that nice speck going on saying put a string here. It's going to return true if it's a PAIL-indrome and false if it's a PAL-indrome. And it says true. Now maybe you're bugged by this. I know you're bugged by my bad humor, but too bad. Maybe you're bugged by this, saying wait a minute, how does this thing stop? This is the kind of definition that your high school geometry teacher would have rapped your knuckles over. You can't define things in terms of themselves. This is an inductive

definition. Actually we could prove inductively that it holds, but how do we know it stops? Well notice what the computation is doing. it's looking first to see am I in the base case, which I'm done. If I'm not I'm just going to reduce this to a smaller computation. And as long as that smaller computation reduces to another smaller computation, eventually I ought to get to the place where I'm down in that base case.

And to see that I've written another version of this, which I'm going to use here, where I'm going to give it a little indentation. I'm going to call this palindrome 1. Sorry about that. Palindrome 1. I'm going to give it a little indentation so that we can see this. OK. Code is right here. And all it's doing is when I'm getting into the different places I'm simply printing out information about where I am. What I want you to see is notice what happened here. OK. I'm calling palindrome with that. It first calls it on that problem. And the code over here says, OK gee, if I'm in the base case do something. I'm not, so come down here check that the two end points a and a are the same and call this again also. Notice what happens. There's this nice unwrapping of the problem. I just doubled the indentation each time so you can see it. So each successive call, notice what's happening. The argument is getting reduced. And we're going another level in. When we get down to this point, we're calling it with just a string of length one. At that point we're in the base case and we can unwrap this computation. We say, ah, that's now true. So I can return true here. Given that that's true and I already checked the two end points, that's true, that's true. And I unwrap the computation to get back.

You are going to have to go play with this. Rock it if you like to try and see where it goes. But I want to stress again, as long as I do the base case right and my inductive or recursive step reduces it to a smaller version of the same problem, the code will in fact converge and give me out an answer. All right, I want to show you one last example of using recursion because we're going to come back to this. This is a classic example of using recursion. And that is dating from the 1200s and it is due to Fibonacci. Does anyone know the history of what Fibonacci was trying to do? Sorry, let me re-ask that. Fibonacci. Which actually is son of Bonacci which is the name of his father who was apparently a very friendly guy. First of all, does anyone know what a Fibonacci number is? Wow.

STUDENT: [INAUDIBLE]

PROFESSOR ERIC GRIMSON: Right, we're going to do that in a second, but the answer is Fibonacci numbers, we define the first two. Which are both defined to be, or I can define them in multiple ways, and 1. And then the next Fibonacci number is the sum of the previous two. And the next number is the sum of the previous two. Do you know the history of this?

STUDENT: [INAUDIBLE].

PROFESSOR ERIC GRIMSON: Exactly. Thank you. Bad throw, I'm playing for the Yankees. Sorry John. The answer is Fibonacci actually was actually trying to count rabbits back in the 1200s. The idea was that rabbits could mate after a month, at age one month. And so he said, if you start off with a male and a female, at the end of one month they have an offspring. Let's assume they have two offspring. At the end of the next month let's assume those offspring have offspring. Again a male and female. The question was how many rabbits do you have at the end of a year? At the end of two years? At the end of more than that number of years, and so. We can do

this with the following level definition. We're going to let pairs of 0, the number of pairs at month 0, actually it would not be it would be 1. We let the number of pairs at month 1 be 1. And then the number of pairs at month n is the number of pairs at month n - 1 plus the number of pairs at month n - 2. The sum of the previous two. If I write Fibonacci, you see it right there. And the reason I want to show you this is to notice that the recursion can be doubled.

So this says, given a value x, if it's either or 1, either of those two cases, just return 1. Otherwise break this down into two versions of a simpler problem. Fib of x - 1 and fib of x - 2, and then take the sum of those and return that as the value. Notice if I'm going to have two different sub problems I need to have two base cases here to catch this. And if I only had one it would error out. And as a consequence, I can go off and ask about rabbits. Let's see. At the end of 12 months, not so bad. At the end of two years, we're not looking so good. At the end of three years, we are now in Australia. Overrun with rabbits. In fact I don't think the thing ever comes back, so I'm going to stop it because it really gets hung up here. And I'm going to restart it.

What's the point of this? Again, now that I can think about things recursively, I can similarly break things down into simpler versions of the same problem. It could be one version. It could be multiple versions. And we're going to come back throughout the term to think about how to code programs that reflect this. The last point I want to make to you is, you've started writing programs that you would think of as being inherently iterative. They're running through a loop. It's a common way of thinking about problems. Some problems are naturally tackled that way. There are other problems that are much more naturally thought of in a recursive fashion. And I would suggest palindrome as a great example of that. That's easy to think about recursively. It's much harder to think about iteratively. And you want to get into the habit of deciding which is the right one for you to use. And with that, we'll see you next time.