ERIC GRIMSON: Ladies and gentlemen, I'd like to get started. My name's Eric Grimson. I have the privilege of serving as MIT'S chancellor for academic advancement, you can go look up what that means, and like John I'm a former head of course six. This term, with Ana and John, I'm going to be splitting the lectures, so I'm up to date.

OK last time Ana introduced the first of the compound data types, tuples and lists. She showed lots of ways of manipulating them, lots of built in things for manipulating those structures. And the key difference between the two of them was that tuples were immutable, meaning you could not change them, lists were mutable, they could be changed, or mutated. And that led to both some nice power and some opportunities for challenges. And, in particular, she showed you things like aliasing, where you could have two names pointing to the same list structure, and because of that, you could change the contents of one, it would change the appearance of the contents of the other, and that leads to some nice challenges. So the side effects of mutability are one of the things you're going to see, both as a plus and minus, as we go through the course.

Today we're going to take a different direction for a little while, we're going to talk about recursion. It Is a powerful and wonderful tool for solving computational problems. We're then going to look at another kind of compound data structure, a dictionary, which is also mutable. And then we're going to put the two pieces together and show how together they actually give you a lot of power for solving some really neat problems very effectively. But I want to start with recursion. Perhaps one of the most mysterious, at least according to programmer's, concepts in computer science, one that leads to lots of really bad computer science jokes, actually all computer science jokes are bad, but these are particularly bad.

So let's start with the obvious question, what is recursion? If you go to the ultimate source of knowledge, Wikipedia, you get something that says, in essence, recursion is the process of repeating items in a self-similar way. Well that's really helpful, right? But we're going to see that idea because recursion, as we're going to see in a second, is the idea of taking a problem

and reducing it to a smaller version of the same problem, and using that idea to actually tackle a bunch of really interesting problems. But recursion gets used in a lot of places. So it's this idea of using, or repeating, the idea multiple times. So wouldn't it be great if your 3D printer printed 3D printers? And you could just keep doing that all the way along.

Or one that's a little more common, it's actually got a wonderful name, it's called mise en abyme, in art, sometimes referred to as the Droste effect, pictures that have inside them a picture of the picture, which has inside them a picture of the picture, and you get the idea. And of course, one of the things you want to think about in recursion is not to have it go on infinitely. And yes there are even light bulb jokes about recursion, if you can't read it, it says, how many twists does it take to screw in a light bulb? And it says, if it's already screwed in, the answer is 0. Otherwise, twist it once, ask me again, add 1 to my answer. And that's actually a nice description of recursion.

So let's look at it more seriously. What is recursion? I want to describe it both abstractly, or algorithmically, and semantically or, if you like, in terms of programming. Abstractly, this is a great instance of something often called divide-and-conquer, or sometimes called decrease-and-conquer. And the idea of recursion is, I want to take a problem I'm trying to solve and say, how could I reduce it to a simpler version of the same problem, plus some things I know how to do? And then that simpler version, I'm going to reduce it again and keep doing that until I get down to a simple case that I can solve directly. That is how we're going to think about designing solutions to problems.

Semantically, this is typically going to lead to the case where a program, a definition of function, will refer to itself in its body. It will call itself inside its body. Now, if you remember your high school geometry teacher, she probably would wrap your knuckles, which you're not allowed to do, because in things like geometry you can't define something in terms of itself, right? That's not allowed. In recursion, this is OK. Our definition of a procedure can in its body call itself, so long as I have what I call a base case, a way of stopping that unwinding of the problems, when I get to something I can solve directly. And so what we're going to do is avoid infinite recursion by ensuring that we have at least one or more base cases that are easy to solve. And then the basic idea is I just want to solve the same problem on some simpler input with the idea of using that solution to solve the larger problem.

OK, let's look at an example, and to set the stage I'm going to go back to something you've been doing, iterative algorithms. For loops, while loops, they naturally lead to what we would

call iterative algorithms, and these algorithms can be described as being captured by a set of state variables, meaning one or more variables that tell us exactly the state of the computation. That's a lot of words, let's look at an example. I know it's trivial, but bear with me.

Suppose I want to do integer multiplication, multiply two integers together, and all I have available to me is addition. So a times b is the same as adding a to itself b times. If I'm thinking about this iteratively, I could capture this computation with two state variables. One we'd just call the iteration number, and it would be something, for example, that starts at b, and each time through the loop reduces 1. One. And it will keep doing that until I've counted down b times, and I get down to 0.

And at the same time, I would have some value of the computation, I might call it result, which starts at 0, first time through adds an a, next time through adds an a, and it just keeps track of how many things have I added up, until I get done. And yeah, I know you could just do mult, but this is trying to get this idea of, how would I do this iteratively. So I might start off with i, saying there are b things still to add, and the result is 1. The first time through the loop, I add an a, reduce i by 1. Next time through the loop, I add in another a, reduce i by 1, and you get the idea. I just walk down it until, eventually, I got to the end of this computation.

So we could write code for this, and, actually, it should be pretty straightforward. There it is. Going to call it mult_iter, takes in two arguments a and b, and I'm going to capture exactly that process. So notice what I do, I set up result internally as just a little variable I'm going to use to accumulate things. And then, there is the iteration, as long as b is greater than 0 what do I do? Add a to result, store it away, reduce b by 1, and I'll keep doing that until b gets down to being equal to 0, in which case I just return the result. OK, simple solution.

Now, let's think about this a different way. A times b is just adding a to itself b times, and that's the same as a plus adding a to a itself b minus 1 times. OK, that sounds like leisure to me, that sounds like just playing with words. But it's really important, because what is this? Ah, that's just a times b minus 1, by the definition of the top point. And I know you're totally impressed, but this is actually really cool, because what have I done? I've taken one problem, this one up here, and I've reduced it to a simpler version of the same problem, plus some things I know how to do. And how would I solve this? Same trick, that's a times a times b minus 2, I would just unwrap it one more time, and I would just keep doing that until I get down to something I can solve directly, a base case. And that's easy, when b equal to 1, the answer is just a. Or I could do when b is equal to 0 the answer is just 0. And there's code to capture that.

Different form, wonderful compact description, what does it say? It says, if I'm at the base case, if b is equal to 1, the answer is just a. Otherwise, I'm going to solve the same problem with a smaller version and add it to a and return that result. And that's nice, crisp characterization of a problem. Recursive definition that reduces a problem to a simpler version of the same problem. OK, let's look at another example. Classic problem in recursion is to compute factorial, right? n factorial, or n bang if you like, n exclamation point is n times n minus 1, all the way down to 1. So it's the product of all the integers from 1 up to n assuming n is a positive integer.

So we can ask the same question if I wanted to solve this recursively what would the base case be? Well, when n is equal to 1, it's just 1. In the recursive case, will n times n minus 1 all the way down to 1, that's the same as n times n minus 1 factorial. So I can easily write out the base case, and I've got a nice recursive solution to this problem. OK, if you're like me and this is the first time you've seen it, it feels like I've taken your head and twisted it about 180 degrees. I'm going to take it another 180 degrees because you might be saying, well, wait a minute, how do you know it really stops. How do you know it really terminates the computation? So let's look at it.

There is my definition for fact, short for factorial. Fact of 1 is, if n is equal to 1 return 1, otherwise return n times fact of n minus 1. And let's use the tools that Ana talked about, in terms of an environment at a scope, and think about what happens here. So when I read that in or I evaluate that in Python, it creates a definition that binds the name fact to some code, just all of that stuff over here plus the name for the formal parameter, hasn't done anything with it yet.

And then I'm going to evaluate print a fact of 4. Print needs a value, so it has to get the value of fact of 4, and we know what that does. It looks up fact, there it is, it's procedure definition. So it creates a new frame, a new environment, it calls that procedure, and inside that frame the formal parameter for fact is bound to the value passed in. So n is bound to 4. That frame is scoped by this global frame meaning it's going to inherit things in the global frame. And what does it do? It says, inside of this frame evaluate the body of fact. OK, so it says as n equal to 1? Nope, it's not, it's 4. So in that case, go to the else statement and says, oh, return n times fact of n and n as 4, fact of n minus 1 says I need to return 4 times fact of 3.

4 is easy, multiplication is easy, fact of 3, ah yes, I look up fact. Now I'm in this frame, I don't

see fact there, but I go up to that frame. There's the definition for fact, and we're going to do the rest of this a little more quickly, what does that do? It creates a new frame called by fact. And the argument passed in for n is n minus 1, that value, right there, of 3. So 3 is now bound to n. Same game, evaluate the body is n equal to 1? No, so in that case, I'm going to go to the return statement, it says return 3 times fact of 2. And notice it's only looking at this value of n because that's the frame in which I'm in. It never sees that value of n.

OK, aren't you glad I didn't do fact of 400? We've only got two more to go, but you get the idea. Same thing, I need to get fact of 2 is going to call fact again with n bound to 2. Relative that evaluates the body and is not yet equal to 1. That says I'm going to the else clause and return 2 times fact of 1. I call fact again, now with n bound to 1, and, fortunately, now that clause is true, and it says return 1. Whoops, sorry, before I do, so there's the base case. And it may seem apparent to you, but this is important, right? I'm unwinding this till I get to something that can stop the computation. Now I'm simply going to gather the computation up, because it says return 1. Who asked for it? Well that call to fact of 1. So that reduces to return 2 times 1. And who called for that? Fact of 2. That reduces to return a 3 times 2, which reduces to 4 times 6, which reduces to printing out 24. So it unwinds it down to a base case and it stops.

A couple of observations, notice how each recursive call creates its own frame, and as a consequence, there's no confusion about which value of n I'm using. Also notice, in the other frames, n was not changed. We did not mutate it. So we're literally creating a local scope for that recursive call, which is exactly what we want. Also notice how there was a sense of flow of control in computing fact of something, that reduces to returning n times fact of n minus 1, and that creates a new scope. And that will simply keep unwinding until I get to something that can return a value and then I gather all those frames back up. So there's a natural flow of control here. But most importantly, there's no confusion about which variable I'm using when I'm looking for a value of n.

All right, because this is often a place where things get a little confusing, I want to do one more example. But let me first show you side by side the two different versions of factorial. Actually, I have lied slightly, we didn't show this one earlier but there's factorial if I wanted to do it iteratively. I'd set up some initial variable to 1, and then I'd just run through a loop. For example, from 1 up to just below n minus 1, or 1 up to n, multiplying it and putting it back into return product.

Which one do you like more? You can't say neither you have to pick one. Show of hands, how

many of you like this one? Some hesitant ones, how many prefer this one? Yeah, that's my view. I'm biased, but I really like the recursive one. It is crisper to look at, you can see what it's doing. I'm reducing this problem to a simpler version of that problem. Pick your own version but I would argue that the recursive version is more intuitive to understand. From a programmer's perspective, it's actually often more efficient to write, because I don't have to think about interior variables. Depending on the machine, it may not be as efficient when you call it because in the recursive version I've got it set up, that set of frames. And some versions of these languages are actually very efficient about it, some of them a little less so. But given the speed of computers today, who cares as long as it actually just does the computation.

Right, one more example, how do we really know our recursive code works? Well, we just did a simulation but let's look at it one more way. The iterative version, what can I say about it? Well, I know it's going to terminate because b is initially positive, assuming I gave it an appropriate value. It decreases by 1 every time around this loop, at some point it has to get less than 1, it's going to stop. So I can conclude it's always going to terminate. What about the recursive version? Well, if I call it with b equal to one, I'm done. If I call it with b greater than one, again it's going to reduce it by one on the recursive call, which means on each recursive call it's going to reduce and eventually it gets down to a place, assuming I gave it a positive integer, where b is equal to one. So it'll stop, which just good.

What we just did was we used the great tool from math, second best department at MIT. Wow, I didn't even get any hisses on that one, John, all right, and I'm now in trouble with the head of the math department. So now that I got your attention, and yes, all computer science jokes are bad, and mine are really bad, but I'm tenured. You cannot do a damn thing about it.

Let's look at mathematical induction which turns out to be a tool that lets us think about programs in a really nice way. You haven't seen this, here's the idea of mathematical induction. If I want to prove a statement, and we refer to it as being indexed on the integers. In other words, it's some mathematical statement that runs over integers. If I want to prove it's true for all values of those integers, mathematically I'd do it by simply proving it's true for the smallest value of n typically n is equal to 0 or 1, and then I do an interesting thing. I say I need to prove that if it's true for an arbitrary value of n, I'm just going to prove that it's also then true for n plus 1. And if I can do those two things I can then conclude for an infinite number of values of n it's always true.

Then we'll relate it back to programming in a second, but let me show you a simple example of

this, one that you may have seen. If I had the integers from 0 up to n, or even from 1 up to n, I claim that's the same as n times n plus 1 over 2. So 1, 2, 3, that's 6, right. And that's exactly right, 3 times 4, which is divided by 2, which gives me out 6. How would I prove this? Well, by induction? I need to do the simple cases if n is equal to 0, well then this side is just 0. And that's 0 times 1, which is 0 divided by true. So 0 equals 0, it's true.

Now the inductive step. I'm going to assume it's true for some k, I should have picked n, but for some k, and then what I need to show is it's true for k plus 1. Well, there's the left hand side, and I want to show that this is equal to that. And I'm going do it by using exactly this recursive idea, because what do I know, I know that this sum, in here, I'm assuming is true. And so that says that the left hand side, the first portion of it, is just k times k plus 1 over 2, that's the definition of the thing I'm assuming is true. To that I'm going to add k plus 1. Well, you can do the algebra, right? That's k plus 1 all times k over 2 plus 1, which is k plus 2 over 2. Oh cool, it's exactly that. Having done that, I can now conclude this is true for all values of n.

What does it have to do with programming? That's exactly what we're doing when we think about recursive code, right? We're saying, show that it's true for the base case, and then what I'm essentially assuming is that, if it works for values smaller than b, then does the code return the right answer for b? And the answer is, absolutely it does, and I'm using induction to deduce that, in fact, my code does the right thing. Why am I torturing you with this? Because this is the way I want you to think about recursion. When I'm going to break a problem down into a smaller version of the same problem, I can assume that the smaller version gives the answer. All I have to do is make sure that what I combined together gives me out the right result.

OK, you may be wondering what I'm doing with these wonderful high tech toys down here. I want to show you another example of recursion. So far we've seen simple things that have just had one base case, and this is a mythical story called The towers of Hanoi and this story, as I heard it, is there's a temporal somewhere in Hanoi with three tall spikes and 64 jewel-encrusted golden disks all of a different size. They all started out on one spike with the property that they were ordered from smallest down to largest.

And there are priests in this temple who are moving the disks one at a time, one per second, and their goal is to move the entire stack from one spike to another spike. And when they do nirvana is achieved and we all get a really great life. We'll talk separately about how long is this going to take because there's one trick to it. They can never cover a smaller disk with a larger

disk as they're doing it, so they've got a third disk as a temporary thing. And I want to show you how to solve this problem because you're going to write code with my help in a second, or I'm going to write code with your help in a second to solve it.

So let's look at it, so watch carefully, moving a disk of size one, well that's pretty easy, right? Moving a disk of size two, we'll just put this one on the spare one while you move it over so you don't cover it up. That's easy. Moving a disk of size three, you've got be a little more careful, you can't cover up a smaller one with a larger one, so you have to really think about where you're putting it. It would help with these things didn't juggle and there you go, you got it done. All right, you're watching? You've got to do four. To do four, again, you've got to be really careful not to cover things up as you do this. You want to get the bottom one eventually exposed, and so are you going to pull that one over there. If you do the pattern really well, you won't notice if I make a serious mistake as I'm doing this, which I just did. But I'm going to recover from that and do it that way to put this one over here, and that one goes there, and if I did this in Harvard Square I could make money. There you go, right?

OK, got the solution? See how to solve it? Could you write code for this? Eh, maybe not. That's on the quiz, thanks, John, don't tell them on the quiz, damn. All right, I want to claim though that in fact there's a beautiful recursive solution. And here's the way to think about it recursively. I want to move a tower of size n, I'm going to assume I can move smaller towers and then it's really easy. What do I do, I take a stack of size n minus 1, I move it onto the spare one, I move the bottom one over, and then I move a stack of size n minus 1 to there, beautiful, recursive solution. And how do I move the smaller stack? Just the same way, I just unwind it, simple, and, in fact, the code follows exactly that.

OK, I do a little [INAUDIBLE] domain up here to try and get your attention, but notice by doing that what did I do? I asked you to think about it recursively, the recursive solution, when you see it, is in fact very straightforward, and there's the code. Dead trivial, well, that trivial is unfair, but it's very simple. Right? I simply write something, so let me describe it, I need to say how big of tower am I moving and I'm going to label the three stacks a from, a to, and a spare. I have a little procedure that just prints out the move for me, and then what's the solution? If it's just a stack of size one, just print the move, take it to from-- from from to to. Otherwise, move a tower of size n minus 1 from the from spot to the spare spot, then move what's left of tower size one from to two, and then take that thing are stuck on spare and move it over to two, and I'm done.

In that code that we handed out, you'll see this code, you can run it. I'm not going to print it out because, if I did, you are just going to say, OK, it looks like it does the right kind of thing. Look at the code, nice and easy, and that's what we like you to do when you're given a problem. We asked you to think about recursively. How do I solve this with a smaller version of the same problem? And then how do I use that to build the larger solution? This case is a little different. You could argue that this is not really a recursive call here, it's just moving the bottom one, I could have done that directly. But I've got two recursive calls in the body here. I have to move a smaller stack twice. We're going to come back to that in a little bit.

Let me show you one other example of recursion that runs a little bit differently. In this case it's going to have multiple base cases and this is another very old problem, it's called the Fibonacci numbers. It's based on something from several centuries ago when a gentleman, named Leonardo of Pisa, also known as Fibonacci, asked the following challenge. He said, I'm going to put a newborn pair of rabbits, one male and one female, into an enclosure, a pan of some sort. And the rabbits have the following properties, they mate at age one month, so they take a month to mature. After a one month gestation period, they produce another pair of rabbits, a male and a female, and he says I'm going to assume that the rabbits never die. So each month mature females are going to produce another pair. And his question was, how many female rabbits are there at the end of a year, or two years, or three years?

The idea is, I start off with two immature rabbits, after one month they've matured, which means after another month, they will have produced a new pair. After another month, that mature pair has produced another pair, and the immature pair has matured. Which means, after another month, those two mature pairs are going to produce offspring, and that immature pair has matured. And you get the idea, and after several months, you get to Australia. You can also see this is going to be interesting to think about how do you compute this, but what I want you to see is the recursive solution to it. So how could we capture this?

Well here's another way of thinking about it, after the first month, and I know we're going to do this funny thing, we're going to index it 0, so call it month 0. There is 1 female which is immature. After the second month, that female is mature and now pregnant which means after the third month it has produced an offspring. And more generally, that the n-th month, after we get past the first few cases, what do we have? Any female that was there two months ago has produced an offspring. Because it's taken at least one month to mature, if it hasn't already been mature, and then it's going to produce an offspring. And any female that was around last

month is still around because they never die off.

So this is a little different. This is now the number of females at month n is the number of females T month n minus 1, plus the number of females and month n minus 2. So two recursive calls, but with different arguments. Different from towers of Hanoi, where there were two recursive calls, but with the same sized problem. So now I need two base cases, one for when n is equal to 0, one for when n is equal to 1. And then I've got that recursive case, so there's a nice little piece of code. Fibonacci, I'm going to assume x is an integer greater than or equal to 0. I'm going to return Fibonacci of x. And you can see now it says, if either x is equal to 0 or x is equal to 1 I'm going to return 1, otherwise, reduce it to two simpler versions of the problem but with different arguments, and I add them up.

OK, and if we go look at this, we can actually run this, if I can find my code. Which is right there, and I'm just going to, so we can, for example, check it by saying fib of 0. I just hit a bug which I don't see. Let me try it again. I'll try it one more time with fib of 0. Darn, it's wrong, let me try it. I've got two different versions of fib in here, that's what I've got going on. So let me do it again, let's do fib of 1. There we go, fib of 2 which is 2, fib of 3 just three, and fib of 4 which should add the previous two, which gives me 5. There we go. Sorry about that, I had two versions of fib in my file, which is why it complained at me. And which is why you should always read the error instructions because it tells you what you did wrong. Let's go on and look at one more example of doing recursion, and we're going to do dictionaries, and then we're going to pull it all together.

So far we've been doing recursion on numerical things, we can do it on non-numerical things. So a nice way of thinking about this is, how would I tell if a string of characters is a palindrome? Meaning it reads the same backwards and forwards. Probably the most famous palindrome is attributed to Napoleon "Able was I ere I saw Elba." Given that Napoleon was French, I really doubt he said "Able was I ere I saw Elba," but it's a great palindrome. Or another one attributed to Anne Michaels "Are we not drawn we few drawn onward to a new era," reads the same backwards and forwards. It's fun to think about how do you create the palindromes.

I want to write code to solve this. Again, I want to think about it recursively, so here's what I'm going to do. I'm first going to take a string of characters, reduce them all to lowercase, and strip out spaces and punctuation. I just want the characters. And once I got that, I want to say, is that string, that list of characters or that collection of characters as I should say, a

palindrome? And I'm going to think about it recursively, and that's actually pretty easy. If it's either 0 or 1 long, it's a palindrome. Otherwise you could think about having an index at each end of this thing and sort of counting into the middle, but it's much easier to say take the two at the end, if they're the same, then check to see what's left in the middle is a palindrome, and if those two properties are true, I'm done. And notice what I just did I nicely reduced a bigger problem to a slightly smaller problem. It's exactly what I want to do.

OK? So it says to check is this, I'm going to reduce it to just the string of characters, and then I'm going to check if that's a palindrome by pulling those two off and checking to see they're the same, and then checking to see if the middle is itself a palindrome. How would I write it? I'm going to create a procedure up here, isPalindrome. I'm going to have inside of it two internal procedures that do the work for me. The first one is simply going to reduce this to all lowercase with no spaces. And notice what I can do because s is a string of characters. I can use the built in string method lower, so there's that dot notation, s.lower. It says. apply the method lower to a string. I need an open and close per end to actually call that procedure, and that will mutate s to just be all lowercase.

And then I'm going to run a little loop, I'll set up answer or ans to be an empty string, and then, for everything inside that mutated string, I'll simply say, if it's inside this string, if it's a letter, add it into answer. If it's a space or comma or something else I'll ignore it, and when I'm done just return answer, strips it down to lowercase. And then I'm going to pass that into isPal which simply says, if this is either 0 or 1 long, it's a palindrome, returned true. Otherwise, check to see that the first and last element of the string are the same, notice the indexing to get into the last element, and similarly just slice into the string, ignoring the first and last element, and ask is that a palindrome. And then just call it, and that will do it. And again there's a nice example of that in the code I'm not going to run it, I'll let you just go look at it, but it will actually pull out something that checks, is this a palindrome.

Notice again, what I'm doing here. I'm doing divide-and-conquer. I'm taking a problem reducing it, I keep saying this, to a simpler version of the same problem. Keep unwinding it till I get down to something I can solve directly, my base case and I'm done. And that's really the heart of thinking about recursive solutions to problems. I would hope that one of the things I remember, besides my really lousy patter up here, is the idea of Towers of Hanoi, because to me it's one of the nicest examples of a problem that would be hard to solve iteratively, but when you see the recursive solution is pretty straightforward. Keep that in mind as you think

about doing recursion.

OK, let's switch gears, and let's talk very briefly about another kind of data type called a dictionary. And the idea of a dictionary I'm going to motivate with a simple example. There's a quiz coming up on Thursday. I know you don't want to hear that, but there is, which means we're going to be recording grades. And so imagine I wanted to build a little database just to keep track of grades of students. So one of the ways I could do it, I could create a list with the names of the students, I could create another list with their grades, and a third list with the actual subject or course from which they got that great. I keep a separate list for each one of them, keep them of the same length, and in essence, what I'm doing here is I'm storing information at the same index in each list. So Ana, who's going to have to take the class again, gets a B, John, who's created the class, gets an A plus, Sorry Ana, John's had a longer time at it.

All right, bad jokes aside, what I'm doing is I can imagine just creating lists. I could create lists of lists, but a simple way is to do lists where basically at each index I've got associated information. It's a simple way to deal with it. Getting a grade out takes a little bit of work because if I want to get the grade associated with a particular student, what would I do? I would go into the name list and use the method index, which you've seen before, again notice the dot notation it says, this is a list, use the index method, call it on student, and whatever the value of student is, it will find that in the list, return the index at that point, and then I can use that to go in and get the grade in the course and return something out.

Simple way to do it but a little ugly, right, because among other things, I've got things stored in different places in the list. I've got to think about if I'm going to add something to the list I've got to put them in the same spot in the list. I've got to remember to always index using integers which is what we know how to do with lists, at least so far. It would be nice if I had a better way to do it, and that's exactly what a dictionary is going to provide for me. So rather than indexing on integers I'd like to index directly on the item of interest. I'd like to say where's Ana's record and find that in one data structure. And so, whereas a list is indexed by integers, and has elements associated with it, a dictionary is going to combine a key, or if you like, a name of some sort, with an actual value. And we're going to index just by the name or the label as we go into it. So let me show you some examples.

First of all, to create a dictionary I use curly braces, open closed curly brace, so an empty dictionary would be simply that call. If I want to create an actual dictionary, before I insert

things into it, I use a little bit of a funky notation. It is a key or a label, a colon, and then a value, in this case the string Ana and the string b, followed by a comma which separates it from the next pairing of a key and a label, or a key and a value, and so on. So if I do this what it does in my dictionary is it creates pairings of those labels with the values I associated with them. OK, these are pretty simple, but in fact, there's lots of nice things we can do with it.

So once we've got them indexing now is similar to a list but not done by a number, it's done by value. So if that's my key, I can say, what's John's grade, notice the call, it's grades, which is in my dictionary, open close square brackets, with the label John. And what it does, it goes in and finds that in the dictionary, returns the value associated with it. If I ask for something not in the dictionary, it's going to give me a key error. Other things we can do with dictionaries, we can add entries just like we would do with lists. Grades as a dictionary, in open and closed square brackets, I put in a new label and a value, and that adds that to the dictionary.

I can test if something's in the dictionary by simply saying, is this label in grades, and it simply checks all of the labels or the keys for the dictionary to see if it's there, and if it's not returns false. I can remove entries, del, something we've seen before, a very generic thing. It will delete something, and in this case, it says, in the dictionary grades, find the entry associated with that key, sorry, Ana, you're about to be flushed, remove it. She's only getting a b in the class and she teaches it. We've got to do something about this, right? So I can add things, I can delete things, I can test if things are there.

Let me show you a couple of other things about dictionaries. I can ask for all of the keys in the dictionary. Notice the format, there is that dot notation, grades as a dictionary, it says, use the keys method associated with this data structure dictionaries. Open close actually calls it, and it gives me back a collection of all the keys in some arbitrary order. I'm going to use a funny term here which I'm not certain we've seen so far. It returns something we call an iterable, it's like range. Think of it as giving us back the equivalent of a list, it's not actually a list, but it's something we can walk down. Which is exactly why I can then say, is something in a dictionary, because it returns this set of keys, and I can test to see something's in there. I can similarly get all of the values if I wanted to look at them, giving us out two iterables.

Here are the key things to keep in mind about dictionaries. The values can be anything, any type, mutable, immutable. They could be duplicates. That'd actually makes sense, I could have the same value associated, for example, the same grade associated with different people, that's perfectly fine. The values could be lists, they could be other data structures, they

could even be other dictionaries. They can be anything, which is great.

The keys, the first part of it are a little more structure. They need to be unique. Well duh, that make sense. If I have that same key in two places in the dictionary, when I go to look it up, how am I going to know which one I want? So it needs to be unique, and they also need to be immutable, which also makes sense. If I'm storing something in a key in the dictionary, and I can go and change the value of the key, how am I going to remember what I was looking for? So they can only be things like ints, floats, strings, tuples, Booleans. I don't recommend using floats because you need to make sure it's exactly the same float and that's sometimes a little bit challenging, but nonetheless, you can have any immutable type as your key. And notice that there's no order to the keys or the values. They are simply stored arbitrarily by the Python as it puts them in.

So if I compare these two, lists or ordered sequences indexed by integers, I look them up by integer index, and the indices have to have an order as a consequence. Dictionaries are this nice generalization, arbitrarily match keys to values. I simply look up one item by looking up things under the appropriate key. All I require is that the keys have to be immutable.

OK, I want to do two last things I've got seven minutes to go here. I want to show you an example of using dictionaries, and I'm going to do this with a little bit more interesting, I hope, example. I want to analyze song lyrics. Now I'm going to show you, you can already tell the difference between my age and Ana's age. She used Taylor Swift and Justin Bieber. I'm going to use The Beatles. That's more my generation. Most of you have never heard of The Beatles unless you watched *Shining Time Station* where you saw Ringo Starr, right?

OK, what I'm going to do is, I want to write a little set of procedures that record the frequencies of words in a song lyric. So I'm going to match strings, or words, to integers. How many times did that word appear in the song lyric? And then I want to ask, can I easily figure out which words occur most often, and how many times. Then I'm going to gather them together to see what are the most common words in here. And I'm going to do that where I'm going to let a user say, I want every word that appears more than some number of times. It's a simple example, but I want you to see how a mutation of the dictionary gives you a really powerful tool for solving this problem.

So let's write the code to do that. It's also in the handout, here we go. Lyrics to frequency's, lyrics is just a list of words, strings. So I'm going to set up an empty dictionary, there's that

open close curly brace, and here's what I want to do. I'm going to walk through all the words in lyrics. You've seen this before, this is looping over every word in lyrics. Ah, notice what I'm going to do. I'm going to simply say-- so the first part is, I can easily iterate over the list, --but now I'm going to say, if the word is in the dictionary, and because the dictionary is iterable, it's simply going to give me back all of the keys, it's simply going to say, in this case, if it's in the dictionary, it's already there, I've got some value associated with it, get the value out, add 1 to it, put it back in. If it's not already in the dictionary, this is the first time I've seen it, just store it into the dictionary. And when I'm done just return the dictionary. OK?

So I'm going to, if I can do this right with my Python, show you an example of this. I have put in one of the great classic Beatles songs, you might recognize it right there. Mostly because it's got a whole lot of repetitions of things. So she loves you yeah, yeah, yeah, yeah. Sorry, actually they sing it better than I just did it sarcastically. Sorry about that, but I got she loves you there, and here's my code up here, lyrics to frequency. So let's see what happens if we call it. And we say lyrics to frequencies she loves you. And it would help if I can type, all right, we'll try it one more time, lyrics to frequency's, she loves you. Cool, this gave me back a dictionary, you can see the curly braces, and there are all the words that appear in there and the number of times that they appear.

What's the order? You don't care. You don't know. What we want to do is to think about how can we analyze this, so let's go back and look at the last piece of this. Which is, OK, I can convert lyrics to frequencies. So here's the next thing I want to do, how do I find the most common words? Well, here's what I'm going to do, frequencies is the dictionary, something that I just pulled out. So I can use the values method on it which returns and iterable, as I said earlier, again notice the open close because I got to call it.

That gives me back an iterable that has all of the frequencies inside of there, because it's an iterable, I can use max on it, and it will take that editable and give me back the biggest value. I'm going to call that best, I'm going to set up words to be an empty list, and then I'm just going to walk through all of the entries in the dictionary saying, if the value at that entry is equal to best add that entry into words, just append it onto the end of the list. And when I'm done all of that loop, I'm just going to return a tuple of both the collections of words that period that many times and how often they appeared. I'm going to show you an example in a second, but notice I'm simply using the properties of the dictionary.

The last thing I want to do then is say, I want to see how often the words appear. So I'm going

to give it a dictionary and a minimum number of times. And here I'm going to set result up to be an empty list, I'm going to create a flag called false, it's going to keep track of when I'm done. And as long as I'm not yet done, I'll call that previous procedure that's going to give me back the most common words and how often they appeared. I check and remember it was a tuple, how often do they appear, if it's bigger than the thing I'm looking for, I'll add that into my result.

And then the best part is, I'm now going to walk through all the words that appeared that many times, and just delete them from the dictionary. I can mutate the dictionary. And by doing that, I can go back around and do this again, and it will pull out how many times has this appeared and keep doing it. When I can go all the way through that, if I can't find any more, I'll set the flag to true which means it will drop out of here and return the result. I'm going to let you run this yourself, if you do that, you'll find that it comes up with, not surprisingly, I think yeah is the most common one and she loves you, followed by loves and a few others. What I want you to see here is how the dictionary captured the pieces we wanted to.

Very last one, there's Fibonacci, as we called it before. It's actually incredibly inefficient, because if I call it, I have to do all the sub calls until I get down to the base case, which is OK. But notice, every other thing I do here, I've actually computed those values. I'm wasting measures, or wasting time, it's not so bad with fib of 5, but if this is fib of 20, almost everything on the right hand side of this tree I've already computed once. That means fibs very inefficient. I can improve it by using a dictionary, very handy tool. I'm going to call fib not only with a value of n, but a dictionary which initially I'm going to initialized to the base cases. And notice what I do, I'm going to say if I've already computed this, just return the value in the dictionary. If I haven't, go ahead and do the computation, store it in the dictionary at that point, and return the answer.

Different way of thinking about it, and the reason this is really nice is a method called memoization, is if I call fib of 34 the standard way it takes 11 million plus recursive calls to get the answer out. It takes a long time. I've given you some code for it, you can try it and see how long it takes. Using the dictionary to keep track of intermediate values, 65 calls. And if you try it, you'll see the difference in speed as you run this. So dictionaries are valuable, not only for just storing away data, they're valuable on procedure calls when those intermediate values are not going to change. What you're going to see as we go along is we're going to use exactly these ideas, using dictionaries to capture information, but especially using recursion to break

bigger problems down into smaller versions of the same problem, to use that as a tool for solving what turn out to be really complex things. And with that, we'll see you next time.