PROFESSOR: Last lecture we were talking about classes, and object-oriented programming, and we're going to come back to it today. I'm going to remind you, we were talking about it because we suggested it is a really powerful way of structuring systems, and that's really why we want to use it, It's a very common way of structuring systems. So today I'm going to pick up on a bunch of more nuanced, or more complex if you like, ways of leveraging the power of classes. But we're going to see a bunch of examples that are going to give us a sense. I'm going to talk about inheritance, we're going to talk about shadowing, we're going to talk about iterators. But before get to it, I want to start by just highlighting, sort of, what was the point of classes? So I'll remind you.

A class, I said, was basically a template for an abstract data type. And this was really to drive home this idea of modularity. I want the ability to say, I've got a set of things that naturally belong together, I'm going to cluster them together, I want to treat it like it's a primitive, I want to treat it like it's a float or an int or a string. Is this going to be a point or a segment or something different like that. So it's really a way, as I said, of just trying to cluster data together. And this is a notion of modularity slash abstraction where I'm treating them as primitives. But the second thing we talked about is that we also have a set of methods, using the special name method because we're talking classes. But basically functions that are designed to deal with this data structure. We're trying to group those together as well. So we cluster data and methods.

Second key thing we said was, in the ideal case, which unfortunately Python isn't, but we'll come back to that, in the ideal case, we would have data hiding, and by data hiding, which is sort of a version of encapsulation, what we meant was that you could only get to the internal pieces of that data structure through a proscribed method. Proscribed meaning it's something I set up. So data hiding saying, you would only access the parts through a method. And as we said, unfortunately Python does not enforce this. Meaning that I could create one of these data structures, ideally I'd have a method, that I'm going to see some examples of that I used to get the parts out, unfortunately in Python you could take the name the instance dot some internal variable you'll get it back. It is exposed. And this is actually just not a good idea. So I suggested in my very bad humor, that you practice computational hygiene and you only use appropriate methods to get the parts out. OK didn't laugh the joke last time, you're not going to laugh at it this time, I don't blame you. All right, and then the last piece of this is that we said the class is a template. When we call that class, it makes an instance. So class is used to make instances, meaning particular versions, of that structure, and we said inside the instances we have a set of attributes. Internal variables, methods, that are going to belong to that structure.

OK, so with that in mind, here's what I want to do. I'm going to show you a set of examples, and I want to warn you ahead of time, the code handout today is a little longer than normal because we want to build essentially an extended example of a sequence of examples of classes. We're going to see the idea, of which we're gonna talk about, of inheritance or hierarchy, in which we can have classes that are specializations of other classes. We're gonna see how we can inherit methods, how we can shadow methods, how we can use methods in a variety of ways. So this is a way of suggesting you may find it more convenient to put notes on the code handout rather than in your own notes. Do whatever you like, but I just wanted to alert you, we're going to go through a little more code than normal.

So, the little environment I'm going to build is an environment of people. I'll build a simple little simulation of people. So I'm going to start off with the first class, which I've got up on the screen, and it's on your handout as well, which is I'm going to build an instance, or a class rather, of persons. I'm going to draw a diagram, which I'm gonna try and see if I can do well, over here, of the different objects we're going to have. So I've got, a class, and by the way a class is an object. Instances are also objects, but classes are objects. We're gonna see why we want that in a second. Because I'm gonna build an object, sorry a class, called a person. Now, let's walk through some of the pieces here. The first one is, there's something a little different. Remember last time we had that keyword class and then a name, that name, in this case, person says this is the name for the class, and then we would have just had the semicolon and a bunch of internal things. Here I've got something in parens, and I want to stress this is not a variable. All right, this is not a def, this is a class. I'm going to come back to it, but what this is basically saying is that the person class is going to inherit from another class, which in this case is just the built-in Python object class. Hold on to that thought, it's going to make more sense when we look at a little more interesting example, but I want to highlight that. All right now, if we do this, as I said before, we can create a version of a person, let me just call it per, person.

OK? And what we said last time is, when we wanted to create an instance inside of this class definition, we've got one of those built-in things called init. I'm gonna again remind you, some of the methods we have, Underbar underbar init is going to be the thing that creates the instance. Actually slightly misspeaking, actually Python creates the instance, but it's one thing that fills it in. So in this case, I'm going to give it 2 arguments: Frank Foobar Now, you might have said, wait a minute, init here has 3 arguments: self, family name, and first name. So again, just to remind you, what we said happens here is that when I call this class, person, I'm creating an instance. We'll draw a little instance diagram down here. I'm going to give it the name per. And I should have said inside of person, we've got a set of things. We've got our underbar underbar init, we've got, what else do I have up there? Family name. And a bunch of other methods, down to say.

What happens inside of Python is, when we called the class definition, person, it creates an instance, there it is.

Think of it as a pointer to a spot in memory, and then what we do is, we call, or find, that init method, up here, and we apply it. And the first argument self, points to the instance. So this object here is what self looks at. Now you can see what init's going to do. It says, oh, inside of self, which is pointing to here, let me bind a variable, which was, can read that very carefully, it's family underbar name, to the value I passed in, which was 4. Same thing with first name. OK, so the reason I'm stressing this is, self we do not supply explicitly, it is supplied as pointing to the instance, it's giving us that piece of memory. And that is what then gets created. So here's, now, the instance for per. OK, and I put a little label on there, I'm going to call that an isALink, because it is an instance of that class. God bless you.

All right, so once we got this, let's look at what we can do with person. That's why I built person here. And as I said, I've already bound basically, those two pieces. If I want to get a value out, I can give person, or per, rather, this instance, a messaging. In this case I want to get family, what did I say, family name out, now, again I want to stress, what is happening here? per is an instance, it's this thing here. When I say per dot family name, I'm sending it a message, in essence what that does is, it says, from here it's going to go up the chain to this class object and find the appropriate method, which was family name. It is then going to apply that to self, which points to this instance. And that allows it, therefore, is you can see on the code, to look up under self, what's the binding for family name, and print it back up. So self is always going to point to the instance I want and I can use it. OK what else do we have in here? We can get the first name, that's not particularly interesting.

We've got 2 other special methods: that's cmp and str. All right, cmp is our comparison method. And since I, I was about to say I blew it last time, I misspoke last time, a wonderful phrase that politicians like to use, I misspoke last time. Let me clarify again what cmp will do. Underbar underbar cmp is going to be the method you're going to use to compare two instances of an object. Now, let's back up for second. If I wanted to test equality, in fact I could use underbar underbar eq, under under. It's natural to think about an equality tester as returning a Boolean, it's either gonna be true or false, because something's either equal to or not. In many languages, comparisons also return Booleans, which is why I went down this slippery slope. For many languages, either it's greater than or it's not. But Python is different. Python use cmp, in fact it has a built in cmp, which is what we're relying on here. Where am I, right there. And what cmp returns is 1 of 3 values. Given 2 objects, it says if the first one is less than the second one, it returns -1, if it's equal it returns 0, if it's greater than, it returns 1.

So it allows you this broader range of comparisons. And if you think about it, cmp, you could apply on integers, you could apply it on floats, apply it on strings. So it's overloaded, it has the ability to do all of those. And in this case what we're saying is, given 2 objects, let's create a tuple of the first, sorry, family and first name of ourselves, and other is another object, family and first name of that, and then just use cmp to compare them. All right, so it's going to use the base pieces. OK, so it gives me a way of doing comparisons. And str we saw last time as well, this is cmp does comparison, and str is our printed representation.

OK. So what we've got now, is a simple little class. We've also got two methods there. I want to look at them, we're gonna come back to them, but they start to highlight things we can do with our classes. So I've built one simple version of it here, which is per. And notice I've got another method, right up here, called say. And say takes two arguments, for the moment the second argument, or the first argument's, not going to make a lot of sense, but say takes two arguments besides itself. It's going to take another object to which it's saying something and the thing to say. Since I only have one object here, I'm going to have person talk to himself. You may have met a few other undergraduates who have this behavior. I'll have him talk to himself and say, just some random message the faculty members occasionally worry about. OK, what does this thing do? Now you're going to see some of the power of this. Again, remember, I'm down here, I'm sending this the message say, it's going to go up the chain to find the say message in person. And what does say do, it says given another object and some string, it's going to return, oh, and interesting things, part of which you can't see on the screen. First what it does, is it gets first name of self. Remember self is pointing to this instance, so it's simply looks up that binding, which is Frank. It's going to create a string in which it adds to that the family name of self, and then another thing that says to, and then ah, I'm now going to send a message to the other object, saying give me your first name. Going to add that to the second piece, and you can see in this case it happens to be the same first and family name. And then at the end of it, which you can't see here but you can see in your handout, I just append the whole string, so it spits it out.

What's the point of this, other than I can get it to say things? Notice, I can now reference values of the instance. But I can also get values of other instances, by sending in a message. And that's why we have that form right there. And then it glued all together. If you think about this for a second, you might say, wait a minute, actually you might have said wait a minute a while ago, why am I just using the variable name there in the function over here? Well in fact, I could've used the function here, first name open close, right? It would have done the same thing. But because I know I'm inside the instance, it's perfectly reasonable to just look up the value. OK, I could have, although I don't want you to do it, have done the same thing there and used underbar, sorry, first name underbar, sorry, first underbar name, but that's really breaking this contract that I want to happen. I should send the message to get the method back out. So again the standard practices is if you know you're inside the object, you can just access the values. If you're doing it with any other objects, send it a message to get it out.

OK, now, that gives you an ability to say, let's look at one more example here, and then we're going to start building our hierarchy, which is, that this person can also sing. And we've got a little sing method here. And notice what it does, it's going to sing to somebody, I guess you're part of the Chorallaries. You're going to sing something, and notice what it does, it's simply going to use its say method, but add at the end of whatever's being said, just tra la la at the end. So this is now an example of a method using another method. Why would you want that? It's nice modularly. I have one method that's doing saying, I have another method that's just building on it. So if I have is person sing to themselves, not a highly recommended activity, it would help if I had it sing to itself, not sing to sing, sorry about that. Notice what it does. Looks like exactly like a say method, except it's got tra la la at the end. Don't worry I'm not going to sing to you. I'll simply say the words. Power of this, other than the silly examples. You see how I can access variables of the instance, how I can access variables of other instances, going to come back to that, and how I can use versions of my own methods to implement other methods. In this case sing is using say as part of what it wants to get out.

OK, so we got a simple little example. Now, let's start adding some other pieces to this. OK, and what do I want to add. Find my spot here. OK, we're going to add an MIT person. Sorry, machine is -- do this, let's go down. OK so I'm going to add an MIT person. Look at the code for second. Aha! Notice what this says. MIT person says it inherits from person. That is, that's the first thing in parens up there. It says, you know, class of MIT person is person. What that is saying is, that this is a specialization of the person class. Or another way of saying it is, we have a super class, in this case it's person. And we have a subclass, in this case its MIT person. And we're going to walk through some examples, but what it says is that that subclass of MIT person can inherit the attributes of the person class. Can inherit the methods, it can inherit variables.

OK, what does MIT person do? Well, here's 1 of the new things it does. It has a local variable called next id num, which is initially set to 0. See that up there. And then it's got some methods, it's got an init method, a get id method, a few other things. OK, let's run this. In particular, I go back down to this one. Let me just uncomment this and do it here. Assuming my machine will do what I want it to do, which it really doesn't seem to want to do today. Try one more time. Thank you, yep. Still not doing it for me, John. OK, we type it. No idea what Python doesn't like me today, but it doesn't. So we're gonna define p 1, I've lost my keyboard, indeed I have. Try one more time. p 1 MIT person, see how fast I can type here -- OK, now, let's look at what the code does, because again it's going to highlight some things. I called MIT person, push this up slightly, it's going to create an instance down here, I called p 1. And when I would do that, I'm gonna initialize it. So I've got, right up here, an initializer, init for MIT person, takes in the family name and the first name. Notice what it does. Huh. It says, if I'm sitting here at MIT person, I'm going to go up and inherit from person its init function and call it. And what am I calling it on? I'm calling it on self, which is pointing to this object, so I've still got it, and then I'm then going to apply the base initialization. And that does exactly what you'd expect, which is just going to create a binding for family name down here. As well as some other things. So this is an example of inheritance. MIT person inherits the init method from person, can get access to by simply referring to it, and I refer to it right there. And it's take the person class, get its init and apply it to my instance plus those things. So I'm just using the same piece of code

Notice the second thing it does. It says inside of self, I'm going to bind the local variable id name to the value of next id name in MIT person. Self is down here, id num, sorry, not id name. I'm going to bind that to the value that I find my going up to here, which is 0, and having done that, I simply increment that value. OK? So what has this done? It says I now have captured in the class, a local variable that I can keep track of. And when I use it, every time I generate an example, let me build another one. I make p 2 another MIT person. OK, I can do things like saying, what is the id number for each of these. First one is 0, second one is 1, which makes sense, right? I'm just incrementing a global variable. Now, things I want you to see about this. Now that I've got a beginning of a hierarchy, I have this notion of inheritance. I can ask a function inside one class to use a function from a class that it can reach by going up the chain. I just did it there. I can ask it to go get values of variables, right, so that looks good. What else do we have in person or MIT person? Well, we can get the id number, we just did. We have a thing to do with this string. Notice it's going to print out something a little different. In fact, there's a kind of funky form there. Which just says, if I want to print it out, I'm gonna create, what this says to do is, I'm gonna create an output template that has that structure to it, but where I see that percent s I'm going to substitute this value for the first one, that value for the second. So if I say, what is p 1? It says ok, MIT person Fred Smith. On the other hand, if I said, what is per, which is that thing I build earlier, it had a different string method, which is just print out person, those pieces.

All right, one last piece to this and we're going to add to it. Suppose I want Fred to say something. Say something to Jane. OK, he said it. Where's the say method? OK, Fred is an instance of an MIT person. where's the say method? Well, there isn't one there, but again, that's where the hierarchy comes in. Fred is this object here, I'm sending it the message say. That turns into going up the chain to this object, which is the class object, and saying find a say method and apply it to that instance. Fudge-knuckle, it ain't here. Don't worry about it, because it says if I can't find one there, I'm going to go up the chain to this method, sorry to this class, and look for a method there. Which there was one, I have a say method. It's going to use that say method. Apply to it. Well, you might say, OK, what happens if it isn't there? Well, that's where, remember I defined person to be an instance of an object, it will go up the chain one last time to the base object in Python to see is there a method there or not. Probably isn't a say method for an object, so at that point it's going to raise an exception or throw an error. But now you again see this idea that the inheritance lets you capture methods.

Now you might say, why not just put a say method inside of MIT person? Well, if you wanted it to do something different, that would be the right thing to do. But the whole notion here's that I'm capturing modularity, I've got base methods up in my base class. If I just want to use them I'm just going to inherit them by following that chain, if you like, basically up the track. OK, so we've got an MIT person, we can use that. Let's add a little bit more to our hierarchy here. I'm going to create, if I can do this right, a specialization of an MIT person, which is an undergraduate. A special kind of MIT person. All right, so if I go back up here, even though my thing is not going to let me do it, let's build an undergraduate. OK, there's the class definition for an undergrad. We're just starting to see some of the pieces, right, so in an undergraduate, where am I here, an undergraduate. OK, it's also got an initialization function. So if I call undergrad, I'm gonna make an undergrad here, again let me go back down here, line ug 2 it's making undergrad, Jane Doe. Now, what happens when I do the initialization here? Notice what goes on. It simply calls the person initialization method. All right, so I'm down here. I'm going to call the person initialization method, what did do? Sorry, the MIT person method, it calls the person method. Just walking up the chain, that's going to do exactly what I did with all the other ones, so I now have a family name and a first name. So I can, for example, say family name and get it back out. All right?

And then, other things that I can do, well I can set what year the person's in, I can figure out what year they're in, there's this unfortunate overflow error if you've hung around too long, but that's not going to happen to you. And I've now got a say method here, so let's look what happens if I ask the undergraduate to say something. OK, it's not a realistic dialogue I know, but, what did this method do? I asked this object to do a say. And notice what it does. It simply passes it back up to MIT person. There's that inheritance again. It's saying, I'm going to have my base say method say something. I'm going to say it to a person, but all I'm going to do because undergraduates in my experience, at least, are always very polite, I'm going to put "Excuse me but" at the front of it. OK, what am I trying to show you here? I know the jokes are awful, but what am I trying to show you here? That I can simply pass up the chain to get it. In fact, what method does the final say here? What class does it come from? Person class, yes, thank you. It goes all the way up to person, right, because MIT person didn't have a say. So I can simply walk up the chain until I find the method I want to have.

Now this is an example of shadowing. Not a great example, but it's a beginning example of shadowing, in that this same method for an undergraduate, shadows the base say method, it happens to call it, but it changes it. It puts "Excuse me but" at the front, before it goes on to do something. Now again, I could have decided here to actually copy what the original say method did, stitch all the other things together. But again, that loses my modularity. I'd really to only have to change it in one place. So by putting my say method up in person, I can add these nuances to it, and it lets me have something that has that variation. If I decide I want to change what say does, I only have to change it in one place. It is in the person class definition, and everything else will follow through for free.

OK, so now I've got an undergrad, right? Let's look at a couple of variations of what happens here. So first of all, I can -- yes?

PROFESSOR 2: Shadowing here is often sometimes called overriding.

PROFESSOR: Yes, thank you, because I'm going to do a pure example of shadowing in a second, John right. Also called overriding. Part of the reason I like the phrase shadow is, if you think about it as looking at it from this direction, you see this version of init before you see the other ones, or you see that version of say, but it is overriding the base say example. OK, so I can say, what does p 1, sorry, yes, what does undergrad look like? And I said wait a minute, MIT person, not undergrad, is that right? Well, where's the str method? I didn't define one in undergrad, so it again tracks up the chain and finds the str method here, so it's OK undergrads are MIT people most the time, so it's perfectly fine.

OK, now, I have built into this also these cmp methods. So I've got two examples. I've got undergrad, or ug. And then I've got poor old Frank Foobar back there, per person. So suppose I want to compare them? What do you think happens here? Compare sounds weird, right, I compare an undergraduate to a person. I don't know what that's doing, some kind of weird psychological thing, but what do you think happens in terms of the code here if I run this. I know it's a little hard because you got a lot of code to look at. Do I have a cmp method defined somewhere? Yeah. So, it's hard to know what it's going to do, but let's look at it. Hmm. Now sometimes I type things and I got errors I don't expect, this one I did expect. So what happened here? Well let's talk about what happens if I do that comparison I was doing, what was I doing? Ug greater than per? What unwinds into is, I'm going to send to ug, that instance, a cmp method. This is really going to become something like ug dot under under cmp under under applied to per. I think that's close.

What does that do? It says starting in ug, I'm going to look for the first cmp method I could find, which is actually sitting here. I had a cmp method in MIT person. If you look at your code, what does it do? It looks up the id numbers to compare them. Well the, ug has an id number because it was created along this chamber. Remember per over here was just created as a person. It doesn't have an id number, so that's why it complaints. Ok, happens if I do that? Compare per to ug. How many people think I get an error? Wow. How many people think I'm going to get either true or false out of this? A few brave hands. Why? Can I ask you, please? Why do you think I'm going to get a, doesn't matter whether it's true or false, why am I going to have something work this time that didn't work last time?

STUDENT: [INAUDIBLE]

PROFESSOR: Yeah, exactly. And in case you didn't hear it, thank you, great answer, sorry, terrible throw. In this case I'm using per, that's the first part, so it's not symmetric. It's gonna use per to do the look up. And as it was said there, per over here goes up and finds a cmp method here which it can apply. In that case, it simply looked at, remember, it took the tuples of first and last name which are both defined here, and did some comparison on that. So this is a way of again pointing out to you that the things are not always symmetric, and I have to be careful about where do I find the methods as I want to use them.

Ok? All right. Let's add, I'm gonna do two more classes here. Let's add one more class, some people debate whether these are really people or not, but we're going to add a class called a professor. OK. Now what am I doing? I'm creating another version of class down here. Which again is an instance, or a subclass, sorry, not an instance, a subclass of an MIT person. I see that because I built it to be there. Again I've got an initialization that's going to call the person initialization, which we know is going to go up -- I keep saying that -- going to call the MIT person initialization, which is going to go up and call this one. So again I'm going to be able to find names. And I do a couple of other different things here. I'm gonna pass in a rank, full professor, associate professor, assistant professor, which I'm just going to bind locally. But I'm gonna add one other piece here, which is I'm gonna add a little dictionary on teaching. So when I create a professor, I'm gonna associate with it a dictionary that says, what have you been teaching?

And then notice the methods I create. I've got a method here called add teaching, takes, obviously a pointer to the instance. A term, which will just be a string, and a subject. And let's look at what it does right here. OK. In fact the call I'm going to make, I'm not certain I'm going to be able to get away with it, my machine is still wonderfully broken, all right, it is, let me just show you what the calls would look like. As you can see here I'm not going to be able to do them. But I'm going to add teaching, as a method call with this with a string for term, and a subject number. What is this going to do? Yeah, I know I'm just worried if I restart Python, I may not be able to pull the thing back in, so I'm going to try and wing it, John, and see if I can make it happen.

Right, what does that teaching do? It's got one of those try except methods. So what does it say it's going to do? It's going to go into the dictionary associated with teaching, under the value of term, and get out a list. And it's going to append to the end of the list the new subject. So it's going to be stored in there, is then going to be term, and a list of what I taught, in case I teach more than one thing each term. It's going to do that, but notice it's a try. If in fact there is no term currently in the dictionary, started out empty, it's going to throw an error, sorry, not throw an error, it's going to raise an exception. Which is a key error, in which case notice what I'm going to do, I'm not going to treat it as an error. I'm simply going to say, in that case, just start off with an empty, with an initial list with just that subject in and put it in the dictionary. As I add more things in, I'll just keep adding things to this dictionary under that term. And if I want to find out what I'm doing, well I can use get teaching, which says given the term, find the thing in the dictionary under that term and return it. If I get an error, I'm going to raise it, which says there is nothing for that term, and in that case I guess I'm just going to return none.

OK? And then the other two pieces we're going to have here, and we want to look at a little more carefully, I just wanted to show you that example, is a professor can lecture, and a professor can say something. Look at the say method, because this now add one more nuance to what we want to do here. And I think in interest of making this go, let me actually, since I'm not going to get my machine to do this right, let me create a couple of professors. If I look at what that is, it's an MIT person because I didn't have any separate string thing there, and we will create a more important professor. What rank do you want, John? Do you want to stay full?

PROFESSOR 2: Undergraduate.

PROFESSOR: Undergraduate, right, a lot more fun I agree. Sorry about that, and we can again just see what that looks like. And that of course, we'll print out, he's also an MIT person. But now here's what I want to do. I want to say something to my good colleague Professor Guttag. Actually I'm going to start a separate -- I'm going to say something to a smart undergraduate. So if I say, remember we have ug defined as an undergraduate, let me do something a little different here. Well let, me do it that way. It says, I don't understand why you say you were enjoying 6.00. Not a good thing to say, right, but if I say to my good colleague Professor Guttag. I have to spell say right, I know, I need help with this, what can I say? We flatter each other all the time. It's part of what makes us feel good about ourselves. Why is the sky blue? I enjoyed your paper, but why is the sky blue?

OK, terrible examples, but what's going on here? One more piece that I want to add. Here's my say method for professor, and now I'm actually taking advantage of to whom I am saying something. Notice again, what does it do? There's the self argument, that's just pointing to the instance of me. I'm passing in another argument, going to call it to who, in one case it was ug, in one case it was Guttag. And then the thing I want to say, ah, look what it does, it says, check the type. And the type is going to take that instance, I had an instance, for example, of a professor down here, and it's going to pick up what type of object it is. So if the type of the person I'm speaking to is undergrad, let's pause for second. Remember I started away back saying we're building abstract data types. Well, here's a great example of how I'm using exactly that, right? I've got int, I've got float, I now have ug, it's a type. So it's says if the object to whom I'm speaking is an undergrad, then use the same method from person where I'm going to put this on the front. On the other hand, if the object to whom I'm speaking is a professor, then I'm going to tag this on the front and use the underlying say method. On the other hand, if I'm speaking to somebody else, I'm just going to go lecture. All right, and when a professor lectures, they just put it's obvious on the end of things, as you may have noticed.

What's the point I want you to see here? I'm now using the instances to help me to find what the code should do. I'm looking at the type. If the type is this, do that. If the type is this, do something different, ok? And I can now sort of build those pieces up. OK, I said one more class. Notice what we're doing. I know they're silly examples, but, sorry, they are cleverly designed examples to highlight key points. What I'm trying to do is show you how we have methods inherit methods, how have message shadow methods, how we have methods override methods, how we can use instances as types to define what the method should do.

Let me show you one last class, because I'm gonna have one more piece that we want to use. And the last class is, sort of, once you've got a set of professors, you can have an aggregate of them. And I don't know, if a group of geese are gaggle, I don't know what a set of professors are, John. Flamers? I, you know, we've got to figure out what the right collective noun here is. We're going to call them a faculty for lack of a better term, right? Now the reason I want to show you this example is, this class, notice, it only is going to inherit from object. It actually makes sense. This is going to be a collection of things, but it's not a subclass of a particular kind of person. And what I want the faculty to do, is to be able to gather together a set of faculty. So if I go down here, grab this for second, and pull it down so you can see it. It looks like I'm not going to be able to run this because my machine is broken, but basically I'm gonna define a set of professors, and then I'm gonna create a new class called faculty. There's the definition of it. It's got an init. You can kind of see what it does. It's going to set up an internal variable called names, which is initially an empty list, internal variable called ids, which is empty, an internal variable called members, which is empty, and another special variable called place, which we're going to come back to in a second, initially bound to none.

OK, I've got a method called add which I'm going to use down here to add professors to the course 6 faculty. Here's what I want to add to do. First of all, notice I'm going to check the type. If this is not a professor, I'm gonna raise an error, a type error, it's the wrong type of object to pass in. The second thing I'm gonna do is say, if that's okay, then let me go off and get the id number. Now remember, that's right up here, so I'm asking the instance of the professor to go up and get the id number. And I want to make sure I only have one instance of each professor in my faculty, so if the id number is in the list of ids already, I'm going to raise an error, sorry, raise an exception as well, saying I've got a duplicate id. OK? And the reason that's going to come up is, notice what I do now. Inside of the instant self, I take the variable names and I add to it the family name of the person I just added. OK, notice the form. I'm using the method, there's the parens to get the family name of the person. I'm just adding it to the list. I've got the id number, I've added the ids, and I add the object itself into members. So as I do this, what am I doing? I'm creating a list, actually several lists: a list of ids, a list of the actual instances, and a list of the family names. And as a cost I want to add, that's why I can check and see, is this in here already or not?

Now, the last reason I want to do this is, I want to be able to support things like that. This is now different, right, this instance is a collection. I want to be able to do things like, for all the things in that collection, do something, like print out the family names. And to do that, I need two special forms: iter and next. OK, now let me see if I can say this cleanly. Whenever I use a for, in structure, even if it was on just a normal list you built, what Python is doing is returning an, what is called an iterator. Which is something that we talked earlier. It's keeping track of where are you in the list, and how do I get to the next thing in the list? I'm going to do the same thing here, and I'm going to create it for this particular structure. So this little thing iter, when I call a for something in, one of these instances, it calls iter, and notice what it does. It initializes place to 0. That was that variable I had up there. That's basically saying I'm at the beginning of the list. It's a pointer to the beginning of the list, and it returns self. Just gives me back a pointer to the instance. That now allows me at each step in that loop to call next. And what does next do? Next says, check to see if that value is too long, if it's longer than, for example, the list of names, raise an exception called stop iteration, which the for loop will use to say OK, I'm done. I'm going to break out of the for loop. Otherwise, what am I going to do? I'll increment place by 1, that's going to move me to the next place in the list, and then in this case I'll just return the instance itself, right? Members is a list of instances, place I've incremented by 1, I take 1 off of it, I get to it. So iter and next work together. Iter creates this method, that's going to give you a pointer to the place in the structure, and then next literally walks along the structure giving you the next element and returning elements in turn so you can do something with it.

Right, so now what that says is, I can have classes that just have local variables. I can have classes that get methods from other variables, and I can also have classes that are collections. And I've supported that by adding in this last piece. OK once you have all of that, in principle we could start doing some fun things. So let's see what happens if we try and make all of this go. And let me, since I'm not going to be able to run it, let me simply do it this way. If I have my undergraduate, ug. I can -- sorry, let's not do it that way -- I can have undergraduate say things like -- all right, what did I just do wrong here? Do I not have undergrad defined? I do. Oh, I didn't have Grimson, sorry, it's me, isn't it? Thank you. The undergraduate very politely asks why he didn't understand, you can have the professor respond. Again, it simply puts a different thing into there. On the other hand, if Professor Guttag asks me something about understanding, I say I really like this paper on, you do not understand, it's a deep paper on programming languages 5, I think, John, isn't it? What else can you do with this thing, right? You can have an undergraduate talk to an undergraduate, in which case they're still polite. Or you could have -- sorry, let me do that the other way -- you could also have an undergraduate simply talk to a normal person. All right, but the good news is you know eventually you get it done, and when you're really done you can have the undergraduate be really happy about this, and so she sings to herself.

OK it's a little silly, but notice what we've just illustrated. And this is where I want to pull it together. With a simple set of classes, and the following abilities, an ability to inherit methods from subclasses, sorry from superclasses, that is having this hierarchy of things. I can create a fairly complex kind of interaction. I can take advantage of the types of the objects to help me decide what to do. And if you think about that, I know it sounds very straightforward, but you would do exactly that if you were writing earlier code to deal with some numerical problem. All right, if the thing is an integer, do this, if it's a float, do that, if it's a string, do something else. I'm now giving you exactly the same ability, but the types now can be things that you could create. And what I've also got is now the ability to inherit those methods as they go up the chain. So another way of saying it is, things that you want to come away from here, are, in terms of these classes. We now have this idea of encapsulation. I'm gathering together data that naturally belongs as a unit, and I'm gathering together with it methods that apply to that unit. Just like we would have done with float or int. Ideally, we data hide, we don't happen to do it here, which is too bad.

Basically we've got the idea of encapsulation. The second thing we've got is this idea of inheritance. Inheritance both meaning I can inherit attributes or field values. I can inherit methods by moving up the chain. I can also the shadow or override methods, so that I can specialise. And I do all of that with this nice hierarchy of classes. So what hopefully you've seen, between these two lectures, and we're going to come back to it in some subsequent lectures, is that this is now a different way of just structuring a computational system. Now, you'll also get arguments, polite arguments from faculty members or other experts about which is a better way of doing it. So I'll give you my bias, Professor Guttag will give you his bias next time around. My view, object-oriented system are great when you're trying to model systems that consist of a large number of units that interact in very specific ways. So, modeling a system of people's a great idea. Modeling a system of molecules is probably a great idea. Modeling a system where it is natural to associate things together and where the number of interactions between them is very controlled. These systems work really well. And we'll see some examples of that next week. Thanks.