PROFESSOR: There's so many things that we're going to start with today, one, is we're going to review the quiz. And we'll be real quick on that. Then, we're going to talk about object oriented programming, which is something that you'll probably be more interested in for your problem set. So just going down problem (1): it's false, true, false, false, false.

Does anyone have any questions on this? No. Everyone good with that? Anyone wonder why if something's true? Or why something's false?

AUDIENCE: Why is the second one false?

PROFESSOR: Why is it false?

AUDIENCE: Why?

PROFESSOR: It's true. Because, and this is an English thing, so, or language thing, so it can kind of trip people up. But it's basically saying that there are some problems that you need to use recursion or iteration to solve.

AUDIENCE: Why do you always have to use [INAUDIBLE]?

PROFESSOR: Well, if you have a large number of inputs or the inputs are variable, the point is that there are certain problems that you would need to use recursion or iteration to solve them. There are some that you don't. But this is just asking if there are problems that exist.

AUDIENCE: Can you use brute force?

PROFESSOR: Well, brute force usually involves some sort of iteration. Because you have to iterate

through all the possible solutions.

Any other questions? OK.

So the next one, this is just an exercise in code reading. And we can actually just look at how it runs. If I run like that. Also, if you don't have your quiz, I have them up here, if you want to pick it up. But that's just the output of that code.

Does anyone not see how that works? Or want me to step through it?

Question (3) was the double recurring question. And those are the two answers we're were looking for.

Does anyone not see how that works? Or want to try and walk through it?

The way to tackle this problem is walk through in your head. Let's take the first set of input.

At the top, that's your initial call. Now, the string, s, here, is not less than 1, right. You're going to go to this double recursive call here. So that means you're going to get as you return. And the way I got this at is, I'm just taking s from one to the end.

What this means is that this part is going to execute first. And it's the same thing. Go into the function. This is now s. This is obviously not only one character long. So we're going to have, again, another double recursive call. And it's going to look like this.

Now, I've got this call to contend with.

This string is one character long. So what does it do? It just returns the input. So, this function call is just going to return t. And, then, this function is going to get called with t, again. And we already know what happens when you can pass a t. That means that this whole function here, results in t. Then, we tack on the a.

That means this function call, 'at', returns 'ta'.

And then we pass it into this outer function here. And we can already guess what

this is going to return because for this input it just reversed the characters. It just flipped them. For 'ta', it's going to flip them again. The entire function call here, results in 'at'. And then we just append 'm' to it.

The entire return for this one is 'atm'.

It's tricky, but you've just got to step through the code and step through the functions and look at each step, what the function is getting as input.

This was the coding question.

Most of the questions up until now has been, can you read code and understand what it's doing. This one asked you to actually implement a function.

When you start with these questions you should always start from the specification. So, this function is assuming that we have a list of words in lowercase.

lStr is a string of lowercase letters. All the letters in lStr are unique. And that the return of the function are going to be all the words in word list that have a one to one mapping between the letters in the word and lStr.

In English, let's say that my lStr is raft. Ok. Assuming that I have a fairly complete word list, I'm going to have two words that are going to meet that criteria. I can say there's an r, a, f, t. That's pretty obvious. And then we have a t, f, a, r. That's what we're looking at.

A lot of you did something where you iterated through all the words and then you had another loop inside that tried to find this correspondence.

The way that we solved it-- and there are multiple solutions to this. So if your solution worked, you got full credit. But the solution that we came up with is first, we're going to take the letters in lStr and we're going to sort them. So we're going to have a, f, r, t. Alphabetically. And then for each word in word list, we're going to do the same thing. Raft becomes a, f, r, t. Fart becomes a, f, r, t, as well. Then it becomes just a simple string comparison. And all you have to do is iterate through the word list.

Does that make sense to everyone?

Our solution has a trick. You don't need to use this trick. A lot of people didn't and they got full credit. But this is one way of doing it.

Question (5). This was the one where we asked you to find the problem with this code. Or, rather we asked does this code meet the specification.

When you get a question like that, first thing you should do is actually look at the specification. Then, you should look for what this code needs to do. Because the specification is going to tell you what the function needs to do. If it doesn't say you need to do something then, that means it's undefined. Right? So In those cases, you can do whatever, as long as you meet the specification.

The first requirement is it returns a list of the pointwise sum of the elements. That's the first requirement of the specification. And then it gives you 2 implicit requirements. One is this example, where it says if I'm given two vectors, this is what I expect the return to be. In this case, the vectors are two different lengths.

It's also saying that your vectors are not always going to be the same length. In that case, you take the pointwise sum up to the shorter of the two lists and just tack on the remainder from a longer list. So that's the second requirement.

The third requirement is, if you have two empty lists you're going to return an empty list.

And, finally, your fourth requirement is, does not modify input.

Now, you know the four requirements from the specification.

And. now, you need to look in the code and see if this code matches all those requirements.

The first one. Does it return a list containing the pointwise some of the elements. Well, this is the portion of the code that does that. It looks like it meets that

specification.

Result is going to be the longer of the two vectors. In this case if v1 longer than v2. We set result to v1 and other to v2. That's the shorter.

Then, if we choose the longer, we set result to v2. And other to the shorter of the two vectors.

Does everyone see that?

We iterate through and we get the pointwise sum. We meet the first requirement.

The second requirement is that if we're given two vectors that are different lengths then, we're going to sum up the furthest that we can, up to the length of the shortest one an tack on the remainder. Well, again, this for loop here, that does a pointwise sum, it only goes up to the length of the shorter list. Second requirement met.

Third requirement. Two empty vectors returns an empty vector. Well, if I have an empty vector here, this is going to be 0. This FOR loop is never going to execute. And my result is going to be empty.

And, then, finally that leaves a fourth requirement, does not modify the input. What's result? Result is what we're ultimately returning. And that's the only thing that we really modify in this function. Result in this case, is v1 or v2. But they're aliased. So it's modifying the inputs and that's a violation.

So the answer to this problem is a total of six characters.

Some of you wrote entire redefinitions of the function or, copied Ryan's code from the reviews. Perfectly acceptable, but way too much work. Remember, programmers are lazy. That's all we were looking for.

If you used the code from Ryan or, you had a different implementation that met the specifications, but was completely different you got full credit.

You have a question?

AUDIENCE: [INAUDIBLE] . I'm just trying to figure out what the distinction is [INAUDIBLE] do a dot copy. Why one is better than the other.

PROFESSOR: Dot copy applies only to dictionaries.

AUDIENCE: Oh.

PROFESSOR: So if we try and do a dot copy on a list.

AUDIENCE: Oh, OK.

PROFESSOR: Got it. I think we took one point for that or something. We knew what your intent was but you didn't have your IDE there with you.

Question (6) was another exercise in code reading. The way that I would attack this one is to figure out what the two functions do first. Let's take the easier of the two, addUp. Takes a dictionaries input and it has a variable result that it initially sets to 0. And then it iterates through all the keys in the dictionary and adds them to result. Basically, its assuming that the values in the dictionary are some sort of number, and it's summing them up, and returning the total.

And, then, this f function here -- takes the dictionary it zeroes out any of the keys it might have. And then it iterates through all the characters in s. If the character is already in the dictionary then, it's going to add 1 to it. So it's going to increment it. And if the character isn't in the dictionary, then, it's going to set up to 0.

Then, it's going to return the result in dictionary. Knowing that, the function becomes pretty easy. f of abbc for d1, which is an empty dictionary. Just walking through it, starting from this point, if we iterate a, it's not going to be in the dictionary. So we're going to set d of a to 0.

Then, we move on to the next character, b. b is not in the dictionary. So d of b becomes 0.

Now, we get one of the second b. b Is now in the dictionary. So we increment b. Now, d of b is 1.

Then, we move on to the final character, c, again not the dictionary. So we set d of c to 0.

Then, we return the dictionary.

That means that in my dictionary, I have three keys a, b, and c. And they have values 0, 1, and 0 Respectively. So add up is going to return 1. And same process for all of these.

Question I have here is what happens when Python gets to that line? Anyone. It's going to be an error. Why?

AUDIENCE: Result is a local variable.

PROFESSOR: Right. Result is a local variable to addUP. There you go. So, again, the approach to this problem is to figure out which each of the functions do, and then right walk through the code.

Did anyone have trouble with this or want me to actually step through it?

First, when f gets an integer, it just prints out the integer in binary. And then, this loop here, prints out the binary representations from 0, 1, 2.

Why is the first output none in this case?

Because in that first iteration i is 0. When f is called, n, is going to be 0. It's just going to return nothing. So it gives you nothing.

Now, the next question was under the assumption that the log base 2 is o of n, what is the order of the function, f?

And to figure this out, you know that this function here is o of n, because we told you. We know that that's one of the first things that's called in f. So automatically, a run time is o of n.

Now, this loop here, iterates how many times?

Log n. Well, log base 2n, to be explicit.

For this function, which is the dominating term here?

When we want to see o of n, it's just going to be that.

If you had o of n plus log n, I think we took a point. Just because when we talk about worst case scenario, we're looking for what the dominating portion of this function is.

How does it do it?

You want to walk through the code now? Alright.

First thing it does, is it gets something it's calling curve digit. All that is, is you take the log base 2 of a number you're going to get the number of binary digits in it.

Think of it as like if I have three which binary is 1, 1. If I take log base2 of this, then my curve digit is going to be 1.

I'm not sure, Python's been around too long.

Well, 1.5 but if we truncate it to an int it's going to be 1. That's kind of like our position marker in the binary number.

Now, we're going to iterate while the current digit is greater than or equal to 0. We're basically going to start here and move down the line in this direction. All it does is says if my n mod 2 -- so I'm checking to see if it's odd or even -- if it's going to be 1 or 0.

In this case, it's going to be 1 to the power of the current digit, which in this case is 1. I'm sorry, misspoke. n is 3, The remainder is going to be 1. In this case 1 to the power of 1 is going to be 1. That's less than n, which it is. Then, my ans is going to be ans plus 1. I'm going to add 1 to the string and construct it.

Then, I'm going to subtract whatever this part is. So 2 to the current digit, in this case to the 1. It's just going to subtract this off.

Then, curve digit is going to be decremented and moved here.

Then, in this iteration, curve digit is going to be 0 and n is going to be 1. So 1 mod 2 is going to be 1. Curve digit is 0. So that's going to be 1 less than 1. It should print it out. I was not prepared for that.

All right. So. the final question -- number (8). Big O notation, if we match it up, does anyone know what it is?

AUDIENCE: Upper bound.

PROFESSOR: Yeah. A lot of you put the expected running time. And the letters are messed up on this but-- a lot of you put the expected running time, but when we are talking about Big O, we're talking about worst case scenario. So, that's the upper bound.

There is an expected bound. If you decide to do any more algorithm analysis, you have Big O, expected, and little o. If I plot the run time of a given function, my Big O might be this. The worst time, my expected, might be like that. And my absolute best case might be like that. When we say they go that's what we're looking for.

Alright Newton's method. What is that an example of?

AUDIENCE: [INAUDIBLE]

AUDIENCE: Yeah. These don't look right. You know what, I'm sorry. This is a different version of the quiz.

Newton's method, that's an approximation.

Then the last one was recursion on your test. The answer we were looking for was induction. That's that.

Anyone have any actual questions.

AUDIENCE: Go back to number (4).

PROFESSOR: Number (4).

AUDIENCE: Yeah. [INAUDIBLE]

PROFESSOR: What was the part that you did not understand with number (4)?

AUDIENCE: I thought it was confusing how to join the [INAUDIBLE] together or how to go through them to see if you know exactly [INAUDIBLE].

PROFESSOR: Well, that's kind of the trick we have here. We know we have a list of words so to iterate through the words is just a FOR loop. So that's what that for word in wordList does. To do the thing where you match the letters one to one, what we implemented here is, we first take lStr and we sort it, sort the characters in lStr.

Then, what we do for wordList is for each word, we sort the characters in that word. And what that does is, it allows us to just directly compare the two strings. And if they're equal, then we've met the criteria for adding it into the wordList, Or the return wordList.

That's the quiz. If you don't have it, you can come pick it up.

AUDIENCE: Sorry, I was just going to ask really quick. If you're getting a [INAUDIBLE] string, you're concatenating [INAUDIBLE] strings, the empty string, why can't you just set it equal to the [INAUDIBLE] string? Is there something about the way that operates that you couldn't do it?

PROFESSOR: Yeah, so if I say my string is 'abcdef,' and I just say sorted(s), this returns a list. What I'm doing with join is I'm just converting it back to a string. Got it?

OK, so on to object oriented programming. So what can someone tell me about classes? What do they allow you to do?

AUDIENCE: Allow you to define a custom type.

PROFESSOR: Yes. So one thing they allow you to do is to define a custom type. Now, when you define a class, you can group your methods and data together with something called encapsulation. So first stuff, we've actually already been using classes. We just didn't tell you, right? So ints, floats, dicts, et cetera, these are all types of

classes.

And each of these, we've already seen them, have something called methods associated with them, right? Methods are basically functions that are associated with a given class. So for example, if I have the str class, which everyone's seen, then it has, say, a method, dot lower. So if I have s equal-- well, let me write it up here on the code. If I say something like s equal 'abcdef,' I can call the method, lower. So we've actually already been doing object oriented programming. You just didn't know it.

Along with that, classes have methods. And they also have something else. Someone help me? What?

AUDIENCE: Parameters? Variables?

PROFESSOR: I'm looking for something-- terminology-wise, attributes. So they're a way of grouping methods and attributes. So when we talk about attributes, we're talking about things that pertain to a specific instance of a class. So let's say that I have a real world example. I have a person class. We've already kind of seen this guy.

A person has multiple instances. So there's an instance of Mitch. There's an instance of Garthi. There's an instance of Phillipe. We're all people, mostly human. And we all have attributes.

So we all have an age, when we were born. We all have a name. Some of us have hair and other attributes, right? We also have actions that we can take. So I can talk. And I could walk. I can talk to you, you, you, you, you, and you. So a method that I could define for a person might be talk to someone.

So that's one way of thinking about objects. So object-oriented programming also gives us something called inheritance, right? So I could think of-- if I'm going along with my person analogy-- I could think of sub-classing person if I'm willing to draw hard binary on the genders. You have males and you have females, right? And in this column, I might have Tracy. I'm just picking on people who come to office hours. And then, Garthi, et cetera.

Now, the inheritance portion of it is important because I've already said that one of my methods on a person is I can talk to people. I can talk to other people. It shouldn't matter if I'm talking to Garthi, as a male, versus Tracy, as a female. I talk to people basically the same. So that gets into something called polymorphism, which is we treat objects with a common super class the same as their sub-classes.

So as another example, let's say that I have dogs and-- what's another canine, foxes? I'm not going to necessarily talk to canines the way that I talk to a person. So they would exist with a different super-class. And then, we could all be animals. So really, what object-oriented programming gives you is a different way of thinking about how you're modeling your world. And what I want to do is now, instead of just talking about these abstract things, walk through some concrete examples.

So the first thing that I want to illustrate is let's say that we want to create a person. But we don't want to use object oriented programming. Or, we don't want to use classes. Professor Guttag will get angry at me if I say we're not using object oriented programming. So let's say I have a function, makePerson. And I'm going to represent a person as a dictionary that has name, age, height, weight as keys. All makePerson does is it takes these things as parameters, makes a dictionary with them as values, and then returns a dictionary.

Then, I have a bunch of helper functions, like get_name of person. All that does is it just returns whatever's in the name key. I can also set the name in case a person decides they don't want to be known as Mitch. They want to be known as Mitchell or something like that. So I have a bunch of these getter and setter function. These are called accessor and mutator functions, whatever terminology you want to use.

I can also define another function that will do something like print out the person. So in this case, I'm just going to print out name, age, height, and weight. And to see this in action, I'm going to make a person, Mitch, 32 70, 200. And then, Serena, 25, 65, 130. I don't know if these are actually correct. So don't quote me on it. I have a syntax error. So if I run this, then all it's going to do is just print out what I'd expect it to print out.

I can also set my age. So I can go back in time to 25, which is a great age. And now, I'm 25. Now, this is fine if you just want to do simple things. But the reason why we kind of like object oriented programming is because we run into difficulties.

So let's say that I print out the type of Mitch. It says I'm a dict. But it doesn't give me any more information. So that means that I could define any random old dict and pass it to some of my functions that I've defined to work on people. And it will probably give me an error. It also makes other operations kind of non-intuitive.

So let's say that I want to figure out if two people are equal. Well, I could do that by defining a function, people_equal, or equal_people, and passing in it a person1 and person2. And for our intents and purposes, it's just going to be if they have the same name, they're the same person. So this is going to, of course, do what we expect it to do and return false, right? Because Serena and Mitch aren't the same person.

But it's kind of awkward. And if we do things using classes, it becomes a little bit more elegant. And you get a lot more power. So let's say I do the exact same thing with a class. So I have the class keyword. I have the name of my object, or my class, my new type. And then, I have this thing called init.

All init does is it says, when I get a new person object, Python automatically calls init with whatever parameters are specified and tells the object to make attributes or to set itself up. So in this case-- need a bigger screen-- I'm making a Mitch person, all right? And the way that you make a new object, or an instance of person, is you have the class name. And then, you pass it whatever parameters are specified in init.

Now, behind the scenes, Python will create a chunk of memory. And then, it'll call this init function. And it'll pass a reference to that chunk of memory. So visually, let's say this is your magical memory. Python sees us call person, goes in, grabs a chunk of memory-- this marker sucks-- grabs a chunk of memory and says, this chunk of memory is of type person.

Then, it calls init. init says, basically, Mitch is a reference to this chunk of memory. This is a self parameter. And then, it has the other parameters. And then, in the init method, all we're doing is we're creating new attributes on this chunk of memory. So we're going to have an attribute name. We're going to have an attribute age, height, weight.

And then, we have a bunch of accessors or getters, mutators or setters. Same exact thing as the functions that I showed, that we had when we were trying to do this without classes. The difference is that these are lexically scoped to person. They are methods for an instance of type person. Whereas before, they were just functions that worked on something we called a person. But a person was actually a dict.

So let me just run this. So if I run this, I'm just going to create a person, a Mitch person and a Sarina person. I'm going to print out Mitch. It's the same thing that I had before, right? Except I'm using a class. Now, if I want to use one of the accessors, so I want to get younger again, I can just take my object, Mitch, and I can call the set_age method and pass it in my new age. And I've lost seven years.

All right, now so far, this is just a different way of doing the same thing, right? But if we look at the type of Mitch, I'm a person. So there's now some extra information that we didn't have before when we were using a dict to represent a person. Before, it could've been any dict. We didn't know that it represented a person. The only reason that it represented a person before we used a class was because we had set certain keys to certain values. But other than that, it was just a dict.

Here, this is marked as being a person. And that also gives us some additional kind of power, which we'll see in a little bit. So now, I've done something sneaky. And I've created a way of comparing Mitch and Sarina. Before I had that function, people_equal. Now, I can use the double equal operator, which is something that we're used to. We use it with all the other types. Why don't we use it with people?

The way that we define that is that on this person class, I have to find this function

underbar, underbar, EQ, underbar, underbar, right? The underbar methods-- and underbar, underbar, init is one of them-- have a special significance in Python. So in this case, the underbar, underbar, EQ, underbar, underbar says that if I have an object niche, and I have a double equal, and I have another object, Serena, it's going to look in the Mitch object. And it's going to say, does it have an EQ method? Does it have this method name?

In this case, it does. So it says, OK, this object is capable of comparing itself to another object for equality. And so it calls this method. And self, in this case, is the Mitch object, is this guy. And the other is the Serena object. Sorry?

AUDIENCE: So when it looks for the EQ, does it look for EQ, or does it look for something that has a double equal sign, and then go off from there? Is the underbar, underbar, EQ, underbar, underbar -- is it defined for all?

PROFESSOR: It's defined for all persons.

AUDIENCE: All persons.

PROFESSOR: Yeah. OK, I see what you're saying. Is it defined for objects other than persons, like all objects?

AUDIENCE: Yeah.

PROFESSOR: Well, we can check that out. You notice that person inherits from object. This is where all classes that you define should inherit from. And so object, like any other type-- remember I showed the dir command before? If you do this, you can see everything that's inherent to an object. So in this case, it doesn't have the underbar, underbar, EQ. So it might be something special that Python does. So this is where I'd actually have to look it up to answer your question. So I'm going to punt on that.

AUDIENCE: So it knows somehow to look for EQ specifically?

PROFESSOR: Yeah. When it sees the double equals sign, it knows to look for EQ. The double underbars signify a magical operator or magical method.

AUDIENCE: So when you're calling EQ, you're not formally giving it two formal parameters in the parentheses. So it's presumably taking the first name before the equals and the second name before the equals--

PROFESSOR: We'll get into that. So the question is, are we actually explicitly passing two formal parameters to it? And the answer is yes and no. This is a bit of syntactic sugar for Python, which means that it's a nicer way of writing things. But we could do something like this. Which is totally awkward, but in actuality, this is what Python does in the background. It does this sort of mangling. And I was actually going to show that with something a little bit easier to comprehend.

So I have a method called get_age, an accessor, right? And usually, you call a method by having a reference to the object, dot, and then whatever method name-- in this case, get_age. You can also write it like this, which is, if you think about it, exactly what happened with the double equal operator, right?

Python took your nice, intuitive syntax-- Mitch double equal Sarina-- and totally mangled it and made it something that was completely ugly. And this is how Python would actually see it. So that's why, when you write these methods, you have a self parameter. This self parameter is like this guy. Python says, ooh, I have an object reference. And they're calling a method, get_age? Do I have a get_age method? Yes, I do. Here it is.

OK, now I'm going to take this and make it look ugly. It's a person object. So I'm going to call person.get_age, and I'm going to pass it Mitch, the reference to the object. And that becomes a self parameter. So that's where the self parameter comes from. There's a little bit of mangling that goes on. And it's partially hidden from you. In other languages, it's completely hidden from you. But in Python, you see some of the ugliness.

AUDIENCE: So in this case, how did it know I was talking about Mitch self and not--

PROFESSOR: Serena?

AUDIENCE: Yeah.

PROFESSOR: So the question was how did Python know that the parameter was Mitch. Because-- let me comment this out so we explicitly what I'm talking about. So Mitch is a reference to a person that represents me on a computer, right? Python knows that this object is of type person because it chunked that memory and made it of type person. So when it sees this reference and it knows it's of type person and it sees this dot and get_age, what that automatically tells Python is, hey, this instance of person is trying to call a method.

Does this class have a method, get_age? In this case, it does. So python says, oh, it does. So I'm going to call this method, person.get_age, and pass it Mitch as a self parameter. Does it make sense in some way?

AUDIENCE: So it basically allows you to put in the name of the person instead of the attribute, instead of using the type?

PROFESSOR: Yeah, so you could do this. I don't recommend that you do this. I'm trying to illustrate where the self parameter is coming from. It's much cleaner to write-- than it is to write this. Fewer keystrokes, too. Remember, programmers are lazy. But it'll do the same exact thing. So can I move on? Wow, we've only got three minutes left. I don't think I'm going to be able to get through this.

So my intent with this code was to show inheritance with shapes, because that's the classic. I'm going to define a base class called shape. And I'm going to give it some methods, area, perimeter, EQ, and less than. For my purposes, shapes are equal if they have the same area, and they're less than if the area's less than. So when we're doing logical comparisons, we're comparing areas.

So now, I'm going to define a class, rectangle. It inherits from shape. It's got two sides, or the lengths of two sides. So the area's pretty easy, just a multiplication. And the parameter is just the sum of the four sides.

And then, I'm going to define another shape, circle. It's got a radius. And it does the area and the perimeter in the way that you expect a circle to do the area and the perimeter. And then, I've got square, which is a sub-class of rectangle, because a

square is just a special case, right? And all I'm going to do is when I initialize my square, I'm just going to call the rectangle's init method, and give it the same size twice.

So what this gives us is some inheritance, some nice properties. So first off, when Python gets to this line, what's going to happen? s is of type shape, right? So what is the method for shape?

AUDIENCE: [INAUDIBLE].

PROFESSOR: What's that?

AUDIENCE: What is the method for shape?

PROFESSOR: What does the area method for shape do? Because that's what I'm calling here.

AUDIENCE: It's going to throw an error.

PROFESSOR: So yeah, voila. And the reason is that we haven't implemented it here. This method is a placeholder. It's saying that if I have a shape, it should have an area method. And this is something that you should see on your problem set when you're doing PS5 with trigger.

AUDIENCE: So it doesn't do anything on its own. You can ignore it and just put [INAUDIBLE] for a circle.

PROFESSOR: Yeah.

AUDIENCE: It just reminds you?

PROFESSOR: It reminds you, yeah.

AUDIENCE: It's like commenting.

PROFESSOR: What's that?

AUDIENCE: So it's kind of like commenting?

PROFESSOR: It's like commenting, except that you get a little bit nicer error checking. So it is a way of explicitly saying that if I have something that is sub-classed from shape, then I can be guaranteed that there is an area method. But if I sub-class off it-- so rectangle, square, and circle-- they all have concrete implementation for areas. So this is all going to run fine. And then, I have my equality operator and my less than operator. Those will run fine. So let me-- all right.

The reason why this is useful, and the big idea, is if I have a list of shapes-- circle, square, and a rectangle-- then I can treat them all exactly the same because they all have an area method, which is defined on a shape. They all sub-class from shape. So it'll print all those areas. And then, if I wanted to, I could even-- because I have a list, I can call the sort method. And because I've defined the less than operator, it'll sort the shapes in descending order by area. And that's what this last bit of code does unless I comment out my assignment.

There we go. So I know I rushed through the shape implementation, and I'm sorry. But I'm actually out of time.