ANA BELL: All right, everyone, let's get started. So today is going to be the second lecture on object-oriented programming. So just a quick recap of last time-- on Monday, we saw-- we were introduced to this idea of object-oriented programming, and we saw these things called abstract data types. And these abstract data types we implemented through Python classes. And they allowed us to create our own data types that sort of abstracted a general object of our choosing, right?

So we've used lists before, for example. But with abstract data types, we were able to create objects that were of our own types. We saw the coordinate example. And then at the end of the class, we saw the fraction example.

So today we're going to talk a little bit more about object-oriented programming and classes. We're going to see a few more examples. And we're going to talk about a few other nuances of classes, talk about information hiding and class variables. And in the second half of the lecture, we're going to talk about the idea of inheritance. So we're going to use object-oriented programming to simulate how real life works. So in real life, you have inheritance. And in object-oriented programming, you can also simulate that.

OK, so the first few slides are going to be a little bit of recap just to make sure that everyone's on the same page before I introduce a couple of new concepts related to classes. So recall that when-- in the last lecture, we talked about writing code from two different perspectives, right? The first was from someone who wanted to implement a class. So implementing the class meant defining your own object type.

So you defined the object type when you defined the class. And then you decided what data attributes you wanted to define in your object. So what data makes up the object? What is the object, OK?

In addition to data attributes, we also saw these things called methods. And methods were ways to tell someone how to use your data type. So what are ways that someone can interact

with the data type, OK? So that's from the point of view of someone who wants to write their own object type. So you're implementing a class.

And the other perspective was to write code from the point of view of someone who wanted to use a class that was already written, OK? So this involved creating instances of objects. So you're using the object type. Once you created instances of objects, you were able to do operations on them. So you were able to see what methods whoever implemented the class added. And then, you can use those methods in order to do operations with your instances.

So just looking at the coordinate example we saw last time, a little bit more in detail about what that meant-- so we had a class definition of an object type, which included deciding what the class name was. And the class name basically told Python what type of an object this was, OK? In this case, we decided we wanted to name a coordinate-- we wanted to create a Coordinate object. And the type of this object was therefore going to be a coordinate.

We defined the class in the sort of general way, OK? So we needed a way to be able to access data attributes of any instance. So we use this self variable, OK? And the self variable we used to refer to any instance-- to the data attributes of any instance in a general way without actually having a particular instance in mind, OK?

So whenever we access data attributes, we would say something like self dot to access a data attribute. You'd access the attribute directly with self.x. Or if you wanted to access a method, you would say self, dot, and then the method name-- for example, distance.

And really, the bottom line of the class definition is that your class defines all of the data-- so data attributes-- and all of the methods that are going to be common across all of the instances. So any instance that you create of a particular object type, that instance is going to have this exact same structure, OK? The difference is that every instance's values are going to be different.

So when you're creating instances of classes, you can create more than one instance of the same class. So we can create a Coordinate object here using this syntax right here. So you say the type, and then, whatever values it takes in. And you can create more than one Coordinate object.

Each Coordinate object is going to have different data attributes. Sorry, it's going to have different data attribute values, OK? Every Coordinate object is going to have an x value and a

y value. But the x and y values among different instances are going to vary, OK? So that's the difference between defining a class and looking at a particular instance of a class. So instances have the structure of the class. So for a coordinate, all instances have an x value and a y value. But the actual values are going to vary between the different instances.

OK, so ultimately, why do we want to use object-oriented programming? So, so far, the examples that we've seen were numerical, right-- a coordinate, a fraction. But using object-oriented programming, you can create objects that mimic real life. So if I wanted to create objects of-- an object that defined a cat and an object that defined a rabbit, I could do that with object-oriented programming. I would just have to decide, as a programmer, what data and what methods I'd want to assign to these groups of objects, OK?

So using object-oriented programming, each one of these is considered a different object. And as a different object, I can decide that a cat is going to have a name, an age, and maybe a color associated with it. And these three here, on the right, each one of these rabbits is also an object. And I'm going to decide that I'm going to represent a rabbit by just an age and a color, OK? And with object-oriented programming, using these attributes, I can group these three objects together and these three objects together, OK?

So I'm grouping sets of objects that are going to have the same attributes together. And attributes-- this is also a recap of last time-- come in two forms, right, data attributes and procedural attributes. So the data attributes are basically things that define what the object is. So how do you represent a cat as an object? And it's up to you, as the programmer, to decide how you want to do that.

For a coordinate, it was pretty straightforward. You had an x and a y value. If we're representing something more abstract like an animal, then maybe I would say, well, I'm going to represent an animal by an age and a name, OK? So it's really up to you to decide how you want to represent-- what data attributes you want to represent your object with.

Procedural attributes were also known as methods. And the methods are essentially asking, what can your object do, OK? So how can someone who wants to use your object-- how can someone interact with it? So for a coordinate, we saw that you could find the distance between two coordinates. Maybe for our abstract Animal object, you might have it make a sound, OK, by maybe printing to the screen or something like that.

OK, this slide's also a recap of how to create a class just to make sure everyone's on the same

page before we go on. So we defined a class using this class keyword. And we said, class, the name of the class. So now we're going to create a more abstract Animal class. We're going to see, in the second half of the lecture, what it means to put something else in the parentheses. But for now, we say that an animal is an object in Python. So that means it's going to have all of the properties that any other object in Python has.

And as we're creating this animal, we're going to define how to create an instance of this class. So we say def. And this __init__ was the special method that told Python how to create an object. Inside the parentheses, remember, we have the self, which is a variable that we use to refer to any instance of the class, OK? We don't have a particular instance in mind, we just want to be able to refer to any instance, OK? So we use this self variable.

And then, the second parameter here is going to represent what other data we use to initialize our object with. So in this case, I'm going to say, I'm going to initialize an Animal object with an age, OK? So when I create an animal, I need to give it an age.

Inside the __init__ are any initializations that I want to make. So the first thing is, I'm going to assign an instance variable, age-- so this is going to be the data attribute age-- to be whatever is passed in. And then, I'm also making another assignment here, where I'm assigning the data attribute name to be None originally.

Later on in the code, when I want to create an Animal object, I say the class name. And then I pass it in whatever parameters it takes-- in this case, the age. And I'm assigning it to this instance here, OK?

All right, so now we have this class, Animal. We've done the first part here, which is to initialize the class, right? So we've told Python how to create an object of this type. There's a few other methods here that I've implemented. Next two we call getters, and the two after that we call setters, OK? And getters and setters are very commonly used when implementing a class. So getters essentially return the values of any of the data attributes, OK?

So if you look carefully, get_age() is just returning self.age, and get_name() just returns self.name. So they're very simple methods. Similarly, set_age() and set_name()-- we're going to see what this funny equal sign is doing here in the next couple of slides. But setters do a very similar thing where they're going to set the data attributes to whatever is passed in, OK?

So those are getters and setters. And then, the last thing down here is this __str__ method.

And this __str__ method is used to tell Python how to print an object of this type Animal. So if you didn't have this __str__ method, if you remember from last lecture, what ends up happening is you're going to get some message when you print your object that says, this is an object of type Animal at this memory location, which is very uninformative, right? So you implement this method here, which tells Python how to print an object of this type, OK?

So the big point of this slide is that you should be using getters and setters-- you should be implementing getters and setters for your classes. And we're going to see, in the next couple of slides, why exactly. But basically, they're going to prevent bugs from coming into play later on if someone decides to change implementation.

So we saw how to-- so the previous slide, this slide here, shows the implementation of the Animal class. And here we can see how we can create an instance of this object. So we can say a = Animal(3). So this is going to create a new Animal object with an age of 3. And we can access the object through the variable a.

Dot notation, recall, is a way for you to access data attributes and methods of a class, OK? So you can say a.age later on in your program, and that is allowed. It'll try to access the age data attribute of this particular instance of the class, a. So this is going to give you 3.

However, it's actually not recommended to access data attributes directly. So this is the reason-- so you're going to see in the next slide, the reason-- why we're going to use getters and setters. Instead, you should use the get_age() getter method to get the age of the animal. So this is going to return, also, 3. So these are going to do the same thing.

And the reason why you'd want to use getters and setters is this idea of information hiding, OK? So the whole reason why we're using classes in object-oriented programming is so that you can abstract certain data from the user, OK? One of the things you should be abstracting is these data attributes. So users shouldn't really need to know how a class is implemented. They should just know how to use the class, OK?

So consider the following case. Let's say whoever wrote the Animal class wants to change the implementation. And they've decided they don't want to call the data attribute "age" anymore, they want to call it "years," OK? So when they initialize an animal they say self.years = age. So an animal still gets initialized by its age. And the age gets passed into a data attribute named "years," OK?

Since I'm implementing this class, I want to have a getter, which is going to return self.years. So I'm not returning self.age anymore, because age is no longer the data attribute I'm using. So with this new implementation, if someone was using this implementation and was accessing age directly as-- was accessing the data attribute age directly-- with this new implementation, they'd actually get an error, right? Because this animal that they created using my old implementation no longer has an attribute named "age." And so Python's going to spit out an error saying no attribute found or something like that, OK?

If they were using the getter a.get_age()-- the person who implemented the class re-implemented get_age() to work correctly, right, with their new data attribute, years, as opposed to age-- so if I was using the getter get_age(), I wouldn't have run into the bug, OK? So things to remember-- write getters and setters for your classes. And later on in your code, use getters and setters to prevent bugs and to promote easy to maintain code.

OK, so information hiding is great. But having said that, Python's actually not very great at information hiding, OK? Python allows you to do certain things that you should never be doing. OK. So the first, we've just seen. The first is to access data attributes from outside of the class, OK? So if I were to say a.age, Python allows me to do that without using a getter and setter.

Python also allows you to write to data attributes from outside the class. So if I implemented the class Animal assuming that age was a number, an integer, and all of my methods work as long as age is an integer, but someone decided to be smart and, outside of the class, set age to be infinite as a string, that might cause the code to crash, OK? Python allows you to do that. But now you're breaking the fact that age has to be an integer, right? So now the methods should probably be checking the fact that age is an integer all the time.

The other thing that you're allowed to do is to create data attributes outside of the class definition, OK? So if I wanted to create a new data attribute called "size" for this particular instance, Python also allows me to do that. And I can set it to whatever I want, OK? So Python allows you to do all these things, but it's actually not good style to do any of them. So just don't do it. All right.

So the last thing I want to mention-- the last thing about classes before we go on to inheritance-- is this idea called default arguments. And default arguments are passed into methods. And since methods are functions, you can also pass in different arguments to functions.

So for example, this set_name() method had self. And then, this new name is equal to this empty string here, OK? We haven't seen this before. But this is called a default argument. And you can use the function in one of two ways.

The first way is so we can create a new instance of an Animal type object with this line here, a = Animal(3). And then we can say a.set_name(). So this calls the setter method to set the name. And notice, we've always said that you have to put in parameters for everything other than self, OK? But here we have no parameters passed in.

But that's OK, because newname actually has a default argument, OK? So that tells Python, if no parameter is passed in for this particular formal parameter, then use whatever is up here by default. So if I haven't passed in the parameter a.get_na-- a.set_name(), sorry-- a.sett_name() is going to be setting the name to the empty string, because that's what the default parameter is. So in the next line, when I print a.get_name(), this is just going to print the empty string, OK?

If you do want to pass in a parameter, you can do so as normal. So you can say a = Animal(3), a.set_name(), and then pass in a parameter here. And then, newname is going to be assigned to whatever parameter is passed in like that. Whatever you pass in overrides the default argument, and everything is good. So when I print a.get_name(), this is going to print out the name that you've passed in. Questions about default? Yeah.

AUDIENCE: [INAUDIBLE]

ANA BELL: What if you don't provide a default value for--

AUDIENCE: For newname?

ANA BELL: For newname? If you don't provide a default argument for newname and you do this case here, then that's going to give you an error. So Python's going to say something like, expected one argument, got zero, or something like that. Great question. OK.

All right, so let's move on to this idea of hierarchies, OK? So the great thing about object-oriented programming is that it allows us to add layers of abstraction to our code, all right? So we don't need to know how very, very low-level things are implemented in order to use them. And we can build up our code to be more and more complex as we use up these different abstractions.

So consider every one of these things on this slide as being a separate object, all right? Every one of these things can be considered to be an animal, OK? According to our implementation of an animal, the one thing that an animal has is an age, OK? And that's probably true, right? Every one of these things has an age.

But now I want to build up on this and create separate groups, right? And each one of these separate groups that I create on top of Animal is going to have its own functionality, right? They're going to be a little bit more specific, a little more specialized.

So I can create these three groups now, a cat, a rabbit, and a person group. And for example-- so they're all animals, right? They all have an age. But for example, maybe a person's going to have a list of friends whereas a cat and a rabbit do not. Maybe a cat has a data attribute for the number of lives they have left, right, whereas a person and a rabbit do not, OK?

So you can think of adding these more specialized-- adding functionality to each one of these subgroups, OK? So they're going to be more and more specialized, but all of them retaining the fact that they are animals. So they all have an age, for example. So on top of these, we can add another layer and say that a student is a person and is an animal, OK? But in addition to having an age and maybe also having a list of friends, a student might also have a major or-- they're pretty, so maybe-- their favorite subject in school.

So that's the general idea of hierarchies, OK? So we can sort of abstract the previous slide into this one and say that we have parent classes and child classes, OK? The Animal class is like our parent class. It's the highest-level class.

Inheriting from the Animal class, we have these child classes or subclasses, OK? Whatever an animal can do, a person can do. Whatever an animal can do, a cat can do. And whatever an animal can do, a rabbit can do, OK-- that is, have an age and maybe some really basic functionality, OK? But between person, cat, and rabbit, they're going to be varying wildly as to the kinds of things that they can do, right? But they can all do whatever Animal can do.

So child classes inherit all of the data attributes and all of the methods, or behaviors, that their parent's classes have, OK? But child classes can add more information. Like for example, a person can have a list of friends whereas a general animal will not.

It can add more behavior. Like, maybe a cat can climb trees whereas people and rabbits cannot. Or you can also override behavior. So in the previous one, we had animal, person,

student. So maybe we have, an animal doesn't speak at all, but a person can speak. So that's added functionality to the person.

And maybe a person can only say hello. But then, when we talk to a student, we can override the fact-- override the speak() method of a person and say that a student can say, you know, I have homework, or I need sleep, or something like that, OK? So we have the same speak() method for both person and student, because they can both speak. But student will override the fact that they say hello with something else.

OK, so let's look at some code to put this into perspective. So we have this Animal class, which we've seen before. This is the parent class, OK? It inherits from object, which means that everything that a basic object can do in Python, an animal can do, which is things like binding variables, you know, very low-level things, OK? We've seen the __init__. We've seen the two getters, the setters, and the string method to print an object of type Animal.

All right, now, let's create a subclass of Animal. We'll call it Cat, OK? We create a class named Cat. In parentheses, instead of putting "object," we now put "Animal." And this tells Python that Cat's parent class is Animal. So everything that an animal can do, a cat can do. So that includes all of the attributes, which was age and name, and all of the methods. So all the getters, the setters, the __str__, the __init__, everything that the animal had, now the cat has-- the Cat class has.

In the Cat class, we're going to add two more methods though. The first is speak(). So speak() is going to be a method that's going to just take in the self, OK-- no other parameters. And all it's doing is printing "meow" to the screen-- very simple, OK? So through this speak(), we've added new functionality to the class. So an animal couldn't speak, whereas a cat says "meow."

Additionally, through this __str__ method here, we're overriding the animal __str__, OK? So if we go back to the previous slide, we can see that the animal's __str__ had animal, plus the name, plus the age here whereas the cat's __str__ now says "cat," name, and the age, OK? So this is just how I chose to implement this, OK? So here I've overridden the __str__ method of the Animal class.

Notice that this class doesn't have an __init__, and that's OK. Because Python's actually going to say, well, if there's no __init__ in this particular method-- sorry, in this particular class-- then look to my parents and say, do my parents have an __init__, OK? And if so, use that __init__. So that's actually true for any other methods. So the idea here is, when you have hierarchies,

you have a parent class, you have a child class, you could have a child class to that child class, and so on and so on. So you can have multiple levels of inheritance.

What happens when you create an object that is of type something that's been-- of a type that's the child class of a child class of a child class, right? What happens when you call a method on that object? Well, Python's are going to say, does a method with that name exist in my current class definition? And if so, use that.

But if not, then, look to my parents. Do my parents know how to do that, right? Do my parents have a method for whatever I want to do? If so, use that. If not, look to their parents, and so on and so on. So you're sort of tracing back up your ancestry to figure out if you can do this method or not.

So let's look at a slightly more complicated example. We have a class named Person. It's going to inherit from Animal. Inside this person, I'm going to create my own-- I'm going to create an __init__ method. And the __init__ method is going to do something different than what the animal's __init__ method is doing. It's going to take in self, as usual. And it's going to take in two parameters as opposed to one, a name and an age.

First thing the __init__ method's doing is it's calling the animal's __init__ method. Why am I doing that? Well, I could theoretically initialize the name and the age data attributes that Animal initializes in this method. But I'm using the fact that I've already written code that initializes those two data attributes. So why not just use it, OK?

So here, this says, I'm going to call the class Animal. I'm going to call its __init__ method. And I'm going to leave it up to you to-- not you as the class, but I'm talking as the programs is running-- I'm going to leave it up to you to figure out how to initialize an animal with this particular age and what to name it. So Python says, yep, I know how to do this, so I'm going to go ahead and do that for you. So now it says person is an animal. And I've initialized the age and the name for you.

The next thing I'm doing in the __init__ is I'm going to set the name to whatever name was passed in, OK? So in the __init__, notice, I can do whatever I want, including calling methods. And then, the last thing I'm doing here is I'm going to create a new data attribute for Person, which is a list of friends, OK? So an animal didn't have a list of friends, but a person is going to.

The next four methods here are-- this one's a getter, so it's going to return the list of friends. This is going to append a friend to the end of my list. I want to make a note that I actually didn't write a method to remove friends. So once you get a friend, they're friends for life. But that's OK.

The next method here is speak(), which is going to print "hello" to the screen. And the last method here is going to get the age difference between two people. So that just basically subtracts their age and says it's a five-year age difference, or whatever it is. And down here, I have an __str__ method, which I've overridden from the Animal, which, instead of "animal: name," it's going to say "person: name : age," OK?

So we can run this code. So that's down here. I have an animal person here. So I'm going to run this code. And what did I do? I created a new person. I gave it a name and an age. I created another person, a name and an age. And here I've just run some methods on it, which was get_name(), get_age(), get_name(), and get_age() for each of the two people. So that printed, Jack is 30, Jill is 25.

If I print p1, this is going to use the __str__ method of Person. So it's to print "person:", their name, and then, their age. p1.speak() just says "hello." And then, the age difference between p1 and p2 is just 5. So that's just subtracting and then printing that out to the screen.

OK, so that's my person. Let's add another class. This class is going to be a student, and it's going to be a subclass of Person. Since it's a subclass of Person, it's going to-- a student is going inherit all the attributes of a person, and therefore, all the attributes of an animal.

The __init__ method of a student is going to be a little different from the one of Person. We're going to give it a name, an age, and a major. Notice we're using default arguments here. So if I create a student without giving it a major, the major is going to be set to None originally.

Once again, this line here, Person.__init__(self, name, age), tells Python, hey, you already know how to initialize a person for me with this name and this age. So can you just do that? And Python says, yes, I can do that for you. And so that saves you, maybe, like five lines of code just by calling the __init__ method that you've already written through Person, OK?

So Student has been initialized to be a person. And additionally, we're going to set another data attribute for the student to be the major. And we're going to set the major to be None. The student is going to get this setter here, this setter method, which is going to change the

major to whatever else they want if they want to change it. And then, I'm going to override the speak() method.

So the speak method for the person, recall, just said "hello." A student is going to be a little bit more complex. I'm going to use the fact that someone created this random class, OK? So this is where we can write more interesting code by reusing code that other people have written. So someone wrote a random class that can do cool things with random numbers.

So if I want to use random numbers in my code, I'm going to put this "import random" at the top of my code, which essentially brings in all of the methods from the Random class, one of the methods being this random() method. So random() is a random() method from the Random class. And this essentially gives me a number between 0 and 1, including 0 but not including 1, OK?

So this random number I get here is going to help me write my method for speak(), where it's going to-- with 25% probability, it's either going to say, "I have homework," "I need sleep," "I should eat," or "I'm watching TV," OK? So a student is going to say one of those four things. And the last thing I'm doing down here is overwriting the __str__ method.

So let's look at the code. I'm going to comment this part out, and uncomment the student, and see what we get. OK, so here, I am creating the student. I'm creating one student whose major is CS, name is Alice, and age is 20. s2 is going to be another student-- name-- Beth, age-- 18. And the major is going to be None, because I didn't pass in any major here. So by default, using the default argument, it's going to be None.

If I print s1, s2, that's going to print out these two things over here just by whatever __str__ method does. And then I'm going to get the students to speak. And if I run it multiple times, you can see that it's going to print different things each time. So "I need sleep," "I have homework," "I need sleep," "I have homework," yeah. So every time, it's going to print something different. OK, questions about inheritance in this example? OK.

Last thing we're going to talk about in this class is an idea of-- or in this lecture, is the idea of-- a class variable, OK? So to illustrate this, I'm going to create yet another subclass of my animal called a rabbit. So class variables-- so so far, we've seen-- sorry, let me back up. So so far, we've seen instance variables, right? So things like self.name, self.age, those are all instance variables. So they're variables that are specif-- they are common across all of the instances of the class, right? Every instance of the class has this particular variable. But the

value of the variable is going to be different between all of the different instances.

So class variables are going to be variables whose values are shared between all of the instances in the class. So if one instance of the class modifies this class variable, then, any other instance of the class is going to see the modified value. So it's sort of shared among all of the different instances. So we're going to use class variables to keep track of rabbits.

OK, so we're creating this class, Rabbit. tag = 1. We haven't seen something like this before. So tag is our class variable. Class variables are typically defined inside the class definition but outside of the __init__. So tag is going to be a class variable, and I'm initializing it to 1.

Inside the __init__, this tells us how to create a Rabbit object. So I'm going to give it self as usual, an age, and then two parents. Don't worry about the two parents for now. Inside the __init__-- sorry, inside the __init__-- I'm going to call the __init__ of the animal just to do less work. Python already knows how to initialize an animal for me, so let's do that. So that's going to set the two data attributes, name and age.

I'm going to set the data attributes for parent1, parent2 for a rabbit to be whatever's passed in. And then, this is where I'm going to use this class variable. So I'm creating this data attribute instance variable particular to a specific instance called rid, OK? And I'm assigning this instance variable to the class variable. And I access class variables using not self, but the class name-- so in this case, rabbit.tag.

So initially, tag is going to be 1. And then, the __init__ is going to increment the tag by 1 here, OK? So that means that, from now on, if I create any other instances, the other instances are going to be accessing the updated value of tag instead of being 1.

So let's do a quick drawing to show you what I mean. So let's say I have Rabbit.tag here, OK? So initially, tag is going to be 1, OK? And then I'm going to create a new Rabbit object. So this is as I'm calling the code, OK? So let's say this is a rabbit object-- oh boy, OK-- r1.

You know, I actually googled how to draw a rabbit, but that didn't help at all. OK, so r1 is going to be a new rabbit that we create. Initially, what happens is, when I first create this new rabbit, it's going to access the class variable, which, it's current value is 1. So when I create the rabbit ID-- the rabbit ID, r1.rid-- this is going to get the value 1. And according to the code, after I set the rabbit ID to whatever tag is, I'm going to increment the tag. So this is going to say, OK, now that I've said it, I'm going to go back up here and increment the tag to be 2. OK.

So let's say I create another Rabbit object, OK? All right, there-- that's a sad rabbit, r2. The ID of r2 is going to be what? Well, according to the way we create a new Rabbit object is it's going to access whatever the value of tag is, which is a class variable. It was changed by the previous creation of my rabbit, so now I'm going to access that, right? So the value is going to be 2.

And according to the code, the next thing I do after I create the instance rid is I'm going to increment tag. So I'm incrementing the class variable to be 3, OK? So notice that all of my instances are accessing this shared resource, this shared variable called tag.

So as I'm creating more and more rabbits, they're all going to be incrementing the value of tag, because it's shared among all of the instances. And so this value, this tag class variable, keeps track of how many different instances of a rab-- of how many different instances of rabbits I've created throughout my entire program, OK? So the big idea here is that class variables are shared across all the instances. So they can all modify them. But these rids, right, these instance variables, are only for that particular instance. So r2 can't have access to r1's ID value, nor could change it. But it won't change it across all of the different instances, OK?

So that's how the __init__ method works of Rabbit, OK? So we have these tags that keep track of how many rabbits we've created. We have a couple of getter-- we have some getters here to get all the parents. So now let's add a somewhat more interesting function. Oh, I just want to mention, when I'm getting the rid, I'm actually using this cool zfill() function here, or method, which actually pads the beginning of any number with however many zeros in order to get to that number here. So the number 1 becomes 001 and so on. So it ensures that I have this nice-looking ID type thing that's always three digits long.

So let's try to work with this Rabbit object. Let's define what happens when you add two rabbits together, OK-- in this class, not in the real world. OK. So if I want to use the plus operator between two rabbit instances, I have to implement this __add__ method, OK? So all I'm doing here is I'm returning a new Rabbit object, OK? Whoops, sorry about that.

And let's recall the __init__ method of the rabbit, OK? So when I'm returning a new Rabbit object, I'm returning a new Rabbit object that's going to have an age of 0. Self-- so the Rabbit object I'm calling this method on is going to be the parent of the new rabbit. And other is going to be the other parent of the new rabbit, OK?

So if we look at the code, and I run it, this part here, I'm creating three rabbits, r1, r2, and r3. Notice this class variable is working as expected, because the IDs of each of my rabbits increments as I create more rabbits. So we have 001, 002, 003. If I print r1, and r2, and r3-- that was these three lines over here-- the parents of r1 and r2 are None, because that's just the default-- yes, the default arguments for creating a rabbit.

To add two rabbits together, I use the plus operator between two Rabbit objects. And on the right here, I'm testing rabbit addition. And I can print out the IDs of all my rabbits. And notice that, when I've created this new rabbit, r4, the ID of it still kept incrementing. So now, the ID of the fourth rabbit is 004. And then, when I get r4's parents, they are as we want them to be, so r1 and r2.

The other thing I want to do is to compare two rabbits. So if I want to compare two rabbits, I want to make sure that their parents are the same. So I can compare the first parent of the first rabbit with the first parent of the second rabbit and the second parent of the first rabbit to the second parent of second rabbit or getting the combinations of those two. So that's what these two Booleans are doing.

So these are going to tell me-- these are going to be Boolean values, either True or False. And I'm going to return either they have the same parents of that type or the same parents criss-crossed, OK? So here, notice that I'm actually comparing the IDs of the rabbits as opposed to the Rabbit objects directly, OK? So if, instead of comparing the IDs in here, I was comparing the parents themselves, directly, what would end up happening is this function, this method, eq(), would get called over and over again. Because here, we have parents that are rabbits.

And at some point, the parents of the very, very first rabbits ever created by this program are None. And so when I try to call-- when I try to call the parent one of None, that's going to give me an error, OK, something like an attribute error where None doesn't have this parent attribute, OK? So that's why I'm comparing IDs here, OK? And the code in the lecture here shows you some tests about whether rabbits have the same parents. And I've created new rabbits here, r3 and r4, the addition of those two. And r5 and r6 are going to have the same parents down here-- True-- but r4 and r6 don't, OK?

So just to wrap it up, object-oriented programming is the idea of creating your own collections of data where you can organize the information in a very consistent manner. So every single

type of object that you create of this particular type that you create-- sorry, every object instance of a particular type is going to have the exact same data attributes and the exact same methods, OK? So this really comes back to the idea of decomposition and abstraction in programming. All right, thanks, everyone.