[SQUEAKING]

[RUSTLING]

[CLICKING]

ANA BELL: All right, let's get started. So today's lecture is lecture three out of four on the idea of object-oriented programming and creating our own object types through Python classes. The majority of today's lecture will be on this idea of inheritance. But before we get there, I'd like to do a little bit of a recap of the big things that we've seen already. And along the way, we'll be writing a new data type for something more abstract, an animal class, more abstract than what we've seen before. But then, when we get to the idea of inheritance, we'll build upon this animal class with some more animal objects.

So the big idea behind creating our own data types is that we want to mimic what's going on in real life, right? And in real life, we basically have all of these different objects in the world, right? But these objects can be grouped according to some categories, right? So in this particular slide, I've got six different objects. But the three on the left can kind of be grouped together, right? We know that they are a kind of cat. And as such, we know that all these cats, we can describe them using some common properties and common behaviors.

So for these cats, I would say that all these cats have a name an age and a color associated with them, right? So I know that all cats will therefore generally have a name and age and a color, and then some similar set of behaviors. The items on the right, those three objects, I know they can be categorized together. Let's say that they're wild rabbits.

And let's say that, for the wild rabbits, I don't actually give them a name. So I would categorize them, again, using common properties, just an age and a color, no name. And then those three objects on the right also have a common set of behaviors different than the objects on the left. And so we're trying to mimic the idea of these categorizations and data types that we see in real life.

OK, so a little bit of a recap, when we define our own data type in Python, we decide on a bunch of attributes. And attributes can either be data or they can be procedures. The data is, you think of them as what variables make up your object. And this is something that you decide. So for a coordinate object, we've seen this example a lot of times, we decided on x and y values. For this more abstract idea of an animal, well, we can just say that we can describe an animal by its age, so how long it's been alive since birth.

In terms of procedural attributes, these we implemented using methods in Python classes. And the idea behind these is just, how can somebody or somebody who's creating an object of this type manipulate the object? What are some ways to interface with this object? So our coordinate class, one of the more interesting things was to find a distance between a coordinate and another coordinate. But some of the simpler things were to just get the value of the x-coordinate, the y-coordinate, and things like that.

For our animal class that we're implementing today, it's going to be a little bit more abstract. But one of the simplest things is to just say, hey, tell me how long you've been alive. That's basically just grabbing the value of the attribute, the H. So here, we're defining our data object, right? The class keyword tells Python we're creating our new data type. This is the name of our data type. So the type of this thing that we're creating is Animal. In parentheses here, animal's parent is the generic Python object.

And later in today's lecture, we're going to see what happens when we put something else in those parentheses. So the parent of a class that we create is something other than just the generic Python object. And then the very first method that we always write in our new object definition is the init method. This tells Python, how do you create an object of this type, a very basic information that Python needs to know. So the init method is a special dunder method, double underscore, init, double underscore.

And by now, you're familiar. The first parameter of every single method that we define inside a class is this thing called self. And remember, self is a variable, right? It's a variable name that allows us to talk about an object without having created one yet because all we're doing here is defining the class, right? We don't have actual objects created. And so this method here, the init method and all the other methods, are run on an object of type animal. But we don't have that object yet.

So the first parameter will be that object in this abstract sort of way. And then you can put other parameters in that list. And so we say that, when we create a new animal object, we're going to initialize it by its age, so some number. Within the init, what do we usually do? Well, we usually initialize all the data attributes, also called instance variables. So here, how many data attributes do I have for the animal class? Two, exactly, yeah, two.

The first one, self.age, is a data attribute. And we know it's a data attribute because we have that self appearing again, right? If it was just a variable name, like age or years or time or something like that, it would just be a regular old variable. And as soon as that init method ended, that variable would go away. But the fact that we've initialized this variable using self. tells Python, hey, this is a variable that I want to persist for as long as this object exists in memory. So it's an instance variable.

So self.age equals age will create this data attribute, age, and assign it to the parameter passed in age. Now, self.name is also a data attribute. It's just not being passed in the parameter list. And that's OK. Not everything has to be passed into the parameter list. So here, what we're effectively doing is saying, when we create a new animal object, we have to tell it the age, how long it's been alive for. But then the name data attribute is always going to be none. So there's an absence of a value for the name for every animal we create right off the bat.

OK, everyone OK with me so far here?

AUDIENCE: What's the purpose of defining self.name equals None?

ANA BELL: What's the purpose of defining this to be none? Well, later on, I'm going to add some methods that allow you to give a name to an animal if you'd like. But again, this is a design choice that I made. So yeah, you might not make the same design choices. So that's the definition for my class, just these four lines of code. And then down here, we saw, in the past couple lectures, how to create actual new objects, right? So this is where the action actually happens, right?

So here, I'm creating a new animal object. A variable, myanimal, is bound to that animal object, right? So that's my variable name. You can name it anything you want. And then you're telling Python to create a new animal object simply by invoking the name of the class, and then passing in all the parameters that we're expecting here, except for self because self becomes this thing. If I were to draw a box around Animal(3), that is the self. That is this object that I just created.

OK, so that's the init method. Last lecture, we saw some dunder methods. And I think I said, probably the second method you'd ever want to implement for a new class is this dunder str method. Now, the dunder str method tells Python how Python should print an object of type animal, right? Because initially, right off the bat, if we didn't implement this dunder str method, Python would default to the str method of the generic Python object, which just tells us the memory location this object has been created at, which is not very useful.

When we print an animal object, and again, my design choice is to say I'm going to print animal, colon, the name of that animal, colon, and the age of that animal, again, my design choice, right? And remember, the dunder str method returns the string you want to print out. It doesn't print it out straight up. Everyone OK with that so far? It should be review.

OK, so then the other things that we want to include in our class, and this is something that I included no matter what the language you're working with, is these things called getters and setters. So getters are these two right here. Getters are basically very simple functions that return the values of the data attributes that this object has. This object just has two, an age and a name, because they were defined using self.age and self.name.

So here's a getter to just tell me the value of self.age. So all it does is return self.age. And name, all it does is return self.name, very, very simple. Setters, same idea, except that now, we're allowing someone using our class to set the values of these data attributes through these methods, right? So here, all it's doing is taking in a parameter for the thing you want to change the age or the name to, right?

And all it does is say, well, self.age is going to be equal to the thing you passed in, OK? That's the age. So we're changing this to a different number. And then the set name is changing the name data attribute to a different string. And here, I'm using this default parameter that we talked about way back when we talked about functions, right? So if you don't pass in an actual string value, we'll default to the empty string.

So let me show you how this works. So this is my animal class exactly as in the slides. I've got my init str and my two getters and setters. And then I've got two animals being created here, right? So here's a print for animal with age four. And here's a print of animal with age six. So if I run these, it should print animal, colon, none, because I didn't set the name to anything for these two, and then their respective ages, right?

So this is using the str method on a. And this is using the str method on b. OK, and then, we can access, of course, using dot notation, all of our data attributes. So here, I'm accessing the age directly. But since the getter, get_age, just returns for me the value of that data attribute, these will actually print the same thing. So I'm just going to comment these out. So if I'm accessing a's age through either the data attribute directly or through the getter method, it will print four for both, pretty straightforward.

And then we can do some things like this. So I can call the set_name method. So here, I'm passing an actual name for it. And then I can print the name or I can use the getter to print the name, right? So if I run that, you'll see, the name has now been changed for object a. Then, if I run the print method on a, then it prints animal, colon, the new name that I just set it to, fluffy. And then the age has been unchanged.

If I run set_name without a parameter, it will revert to that default parameter for the name, which is the empty string. So the new name of my animal a will just be an empty string. So it's just going to be colon with nothing in there, so no space or anything, just nothing. OK, everyone all right so far? Hopefully, a little review.

So we saw that we can actually grab the exact same value for the age by accessing the age data attribute directly using dot notation or our getter that we wrote. One of these is better than the other in terms of style and in terms of good coding practices and in terms of writing code that's easy to read, easy to modify, robust, things like that. The one that is better to use is the one that accesses the method. Both are using dot notation.

But the first one is actually accessing the internals of my class definition, right? In order to know the value of my data attribute as someone who's just using this code for an animal class, I'd have to actually go in and read the init method to know these data attributes that are being initialized. I don't know about you. But I actually-- let's take an example of a list, something we've used a lot. Have you ever gone into the definition of the list class to see the data attributes that are being initialized? I haven't, right?

All we've been doing is working with methods that allow us to make changes to lists, to do operations on lists, and things like that. So the internal workings of the list class is hidden from us. And that's just the way we like it, right? I don't care how the list is actually implemented. And the same thing should happen here, right? I shouldn't care how I implement the animal class. I shouldn't care what instance variables they're using.

So let me show you why. So if someone who's writing the animal class decides in the future that age was a strange variable name to use and they decide to change the variable associated with how long this animal has been alive to, to be years, right? So here, I've got self.years equals age. That's the only change I've made to my animal class.

So I've made the design decision to change this data attribute to be years. And then, of course, since I'm making this class, I need to make sure all my getters and setters and everything still works with this new data attribute. So my get_age will return self.years, right? I'm returning this variable, a data attribute that I've changed. Well, this is the full code. So you can see the changed data attribute here. I'm using self.years equals age.

And then my getter is going to return self.years. And my setter is going to set self.years. Well, this implementation should be hidden from me, somebody who is just trying to create a bunch of animals in their code. So this code down here will work if I use my method, right? Because the method should still work no matter what the data attribute is called, age or years or time or whatever. But if I had code that accessed that data attribute directly, it doesn't work anymore. It throws an error because, surprise, that attribute no longer exists.

So it's much better style and more robust to use only getters and setters, only methods, to make changes and to manipulate the objects. You should never ever really have to use the data attributes, right? Questions about that? OK, good, because that's something that you'll have to keep in mind on the quiz next Monday, not using data attributes.

All right, so having said that, Python does allow you to do a bunch of questionable stuff. So, first of all, it allows you, as we just saw, to access the data attribute of a particular instance that you create. So you create an object. It's a very specific animal with a specific age. You can just use dot notation to access the value of all of these data attributes. Fine, we'll mess ourselves up in the future because maybe this won't work. But it's not so bad.

However, Python also allows you to change the value of a data attribute outside of the class definition. So this is code we write not within the class. It's code we write as somebody who's using the class. So what does this mean? Well, now, I'm going to set the age data attribute to be whatever I want outside the class definition, right? I could even set it to a dictionary if I wanted to.

In this particular case, I'm setting it to a string infinite. But if I do this, then I risk code on this animal class not working further on because maybe they assume that the age is always a number. And so a different method I might run will not work anymore if I happen to set it to the string. And then one other thing Python actually allows you to do is to add data attributes to instances.

So now, the problem with this is that, let's say I create a whole bunch of instances of animals, right? This animal's got age four. This animal's got age six. This animal's got age five. And then one of these animals, I decide to add a new data attribute to it. Like, only one of these instances now has three data attributes associated with it, a name, an age, and now the size. All the other animal instances I've created only have a name and an age associated with them.

Just this one happens to have this extra data attribute. So now, the whole reason why we're creating our own data types was to be consistent. To bundle the specific set of data and specific set of behaviors together flies out the window because now I have one instance that now has this extra data attribute associated with it. And nobody else does, right? So all that consistency has gone out the window. So never ever do any of these outside of the class definition.

It's totally OK to access data attributes while you're defining a class, but not OK to do any of these outside of the class definition, even though Python allows you to do it. OK, so one of the things I wanted to show you in this lecture is something we haven't really seen so far. And that's actually just working with objects that we create. Yes, when we created fractions and coordinates, we just created a whole bunch of objects and then printed the numerators or printed the object or multiplied them together.

But we never actually wrote nice functions that work with objects of our type. So one of the things I wanted to show you is how to do that. So here's a function that creates a dictionary out of a list. So the input here is going to be a list of whatever I want. And the function, what I would like it to do is to pick up from the list only numbers that are non-negative and just integers.

So in this particular case, I would like my function to pick up the 2, the 5, and the 0, ignoring everything else. And I would like to create a dictionary out of these numbers. And what the dictionary should do is map each one of these numbers, so the 2, the 5, and the 0. These would be my keys. And they should be mapped to animal objects with these ages. So that's an animal of age two. And this is an animal with age five. And this should be an animal with age zero, right?

So my key types are ints. And the values associated with the keys, the type should be animal, this object that I just created, all right? So the code is pretty straightforward. We just have a little loop that goes through each element one at a time in my list. That's for n and L. And then I'm just going to do something to the elements that are integers and greater or equal to zero, non-negative. So that will extract only the 2, the 5, and the 0 as we go through the loop over the elements in L.

And then the key line here is this one in red. I'm going to say, this line just adds an entry to my dictionary, right? So this is the syntax for putting something in a dictionary, right? There's no append or plus in a dictionary or anything like that. It's just straight up indexing. The key you want is n, so either a 2, a 5, or a 0. And the value I want to associate with that key is an animal with age whatever this is, 2, 5 or 0, so exactly what I wrote here.

Everyone OK so far? All right, the loop goes through to the end of the list. And then we've created our dictionary. And we're done. As we're writing this code, how would we debug it or how would we check to see that it worked? Well, the instinct is to say, OK, well, let me check to see if this function worked. So here, this line, animals equals animal_dict(L) will run this function. And it runs it on this L. And at the end, it returns a dictionary, something that looks like this.

So our instinct is to just print that return to dictionary. But if we were to print that, and you can actually run the code in Python, if you print that, you get something like this. And that's because Python doesn't dig through elements of dictionaries or even elements of lists to run the print method recursively. It just runs the print method top level. And the problem is, it knows how to print integers just fine. But it doesn't know how to print a dictionary where the values are animal objects.

And so we run into the same problem where, now, the value associated with key 2 is this animal object at that memory location. But how do I know that I didn't screw up my-- I created an animal with age five where it should have been two, right? So the solution, and you'll probably encounter this on the next quiz if you're debugging your code, the solution is to just iterate through the dictionary in such a way that you run that print statement directly on an object of type animal.

Python knows how to do that, right? We told it the str method, right? We have an str method here. So it knows how to run the print directly on an animal object. It just doesn't know how to run the print where the value of a dictionary is an animal object. So let's replace this print of the dictionary with a little loop. It goes through this, the dictionary's items, right? So n is going to be my key. And a is going to be the value associated with that key.

And I've just got the print statement here. So I'm using an F string here that prints key and whatever value that key is with val, whatever value that is, right? So now, the print statement is being run directly on an object of type, animal. And now, the result of this loop will be this, so key 2 with value. And then it uses the print statement on my animal object. Does that make sense? Everyone OK so far?

AUDIENCE: [INAUDIBLE]

ANA BELL: Yeah, exactly. It's converting the stuff in the dictionary with strings because my print statement is being run directly on that object of type animal. And it knows how to do that. I implemented the dunder str. Everyone OK? OK, so let's have you try this. Let's have you write a little code. So this function, it's going to be very similar. We're not making dictionaries. You'll be making a list. But you'll encounter the same problem.

The input here is going to be two lists of the same length. One list has numbers. One list has strings. And what I'd like you to do is create for me a new list. And the new list is going to have animal objects, where you match index by index. So the resulting animal object at index 0 will basically create for me a new animal with age two and named blobfish, right? The animal, object and the resulting list at index 1 will be with age five and name crazyant. And then the animal object at index 2 will be age one and named parafox.

So we're just doing the same thing index by index, where you create a new animal object with the age, this value, one at a time. And you set the name to be this value one at a time, and then return that list. So that should be line 79. OK, who has a start for me? OK, should we call it L3? OK.

AUDIENCE: But then for n in L1, [INAUDIBLE] having it go through L2 of n.

ANA BELL: Yes, but then if you're doing L2 of n, then this should be the index, right? So how do I make this be the index instead of the element directly? Yeah, yeah.

AUDIENCE: Well maybe n of, over the range column?

ANA BELL: Yeah, exactly, right? So instead of looking at the element directly, let's just look at the range. So for i in, range, and then we need to do len. Pick one of the lists because they're the same length. So now, i is 0, 1, 2, 3, 4, 5, all the index values.

AUDIENCE: Well, and you can make L3 [INAUDIBLE] L1 at index i equal to L2 at index i.

ANA BELL: So L1 at index i, so I need to create an animal with that age, right? So let's do this. Age equals L1 at index i, just to save it as a variable. And name equals L2 at index i. Do we agree? So now that I have age and name stored in these variables, how do I make an animal object with that age? Yeah.

AUDIENCE: You could say L3 at i equals-- well, I know you'd call it animal.h.

ANA BELL: Well, the init method creates for me an animal with that age, right? So when we just create a new animal object, we just pass in that age, right? Like, the constructor requires the age of the animal, right? So when we construct a new animal object, we just invoke the name of our animal. Where is it? Here, right? Or-- sorry, our animal type, our animal class. And then we pass in the age that we want to create this animal with, right?

And that, according to the init method, creates self.age to be whatever is passed in, and the name, None. So we're halfway there. We've created an animal object with the age that we want. But the name data attribute for this object is None. Everyone with me so far? So how do we make the name of this animal object be the one that we saved from that L2 list?

AUDIENCE: You can use the setter function.

ANA BELL: Yeah, exactly. We can use the setter function, yeah, set name right here, right? Don't access the attribute directly. But yeah, we can use a setter function. So this created for me that new animal, right? But I need to actually save that animal somehow, right? Because I need to reference it later. So let me do this. a equals animal with that age. And then we run the setter function on this object, a, right? set_name-- it's just a function.

And what name do we want to set it at, this thing here? So name here is this variable that we extracted from the L2 list. Everyone OK so far? So now, what I have is, object a is a variable that's bound to an animal object. The age was set when we first created it. And the name, we just set through the setter function. And now, we should just put it in my list. My list is originally empty, right?

So now, I don't have a bunch of elements to add it to. So let's just append it to L3, like that, right? I mean, theoretically, I could have created an empty list that was three elements long. And then I could do L3.i. But this works, too. And then at the end, let's return L3, right? Questions about this? This is all right?

OK, so if we run it and we just print the list with these animal objects, we run the same problem as that dictionary one, right? You see I've got a bunch of memory locations here? So to test that I did it right, instead of printing the list, let's iterate through our list, through this little for loop, and just run the print method directly on my object, right? So now, if I run that, it should just run the print statement directly on each of these animals, right? So that's correct, I think. Does that make sense? Yeah.

AUDIENCE: What did you change to make it bigger?

ANA BELL: Oh, so instead of printing the list, this thing, I looped through my list and printed the elements.

AUDIENCE: Oh. So that's not in the function.

ANA BELL: That's not in the function, no. That's just, yeah, that's outside. But this is something pretty common that you'll run into. You'll make a list or dictionary or some structure or tuple or something like that with objects of your type. And when you run the print statement directly on that structure, it doesn't go deeper than top level. And so it prints that uninformative stuff. OK, so in this example, we saw that it's better to access the attributes through getters and setters.

So in addition to the init, the str method, writing getters and setters to have a consistent way of accessing and modifying these data attributes is really important. And then you can even impose restrictions, something like, the types have to be this. Or maybe the age can't be a negative number or something like that. And it allows a lot more consistent use of the object.

So now, let's move on to hierarchies, OK? And this is where we're going to talk about inheritance. So there's something like maybe 28 objects on this slide, right? There's the six we encountered at the beginning of this lecture and 22 up there. So there's 28 separate objects on this slide. And all of these objects, we could say, are of type animal, right? Because by our definition, the attributes for an animal is how long they've been alive. And these all are objects that have been alive for some time.

But in addition to having the attribute for how long they've been alive and an unknown name, we can actually then create separate categories, right? And each one of these boxes that I've created is a different subset of animal, right? We'll call it a subclass or a child of an animal class. And that's because they will bring about different data attributes in addition to what an animal's data attributes are.

And they will bring about different behaviors in addition to the behaviors of our really generic animal object, right? So the things a cat can do, a rabbit might not be able to do. And the things a person can do, a cat won't do and a rabbit can't do, right? So they're all animals. But they all are going to have additional data attributes and additional behaviors that are different in these three categories, right? So I might say something like, the cat has a name, an age, and a pattern or a color.

The rabbit, again, I said are wild. So maybe they don't get a name. But they'll have a color or pattern. And then the age, of course, from the animal, people, of course, have the-- the person object has the age that comes from animal. But in addition, they might have a list of friends or something associated, something like that associated with them right, and a list of friends, something a cat doesn't have, something that a rabbit doesn't have. So you see what I mean.

And we can even go further. We can say, well, if I take my person object, I can now sub-categorize that as well and say, well, this is a student class. And then this student class, I would say a student is a person. So all the data attributes and all the behaviors that a person has, the student also has, and, of course, all the animal stuff because a person is an animal. So for example, let's say an animal is a generic object. It doesn't speak. But let's say a person gets the behavior to speak, right? So for speaking, I might just print Hello to the screen or something simple like that.

A student is a person. So maybe they also get something like their age, the name, and maybe a list of friends associated with them. But a student might also have a major or a favorite subject in school associated with them, something that a person doesn't have, right? So that's a new data attribute associated with a student that's not associated with a person. A student might also have different behaviors, like tell me your favorite subject in school, things like that.

Or it might override behaviors of a person. So if a person speaks, says, hello, prints hello to the screen, we can say, hey, if I ask the student to speak, they might say, I have homework instead or something like that, right? So what we're trying to do is take those relationships and implement them in code. So here, I've got an animal class, which is my base class. It's going to be my, also called parent class or super-class. And then anything that an animal has, all the data attributes and all the behaviors of that animal, will be inherited by person, cat, and rabbit.

So anything-- so a person is an animal. A cat is an animal. A rabbit is an animal. So everything they have, all these three subtypes will have as well. But all these subtypes will be different amongst themselves, right? A person will have an ability to speak, maybe print hello to the screen. A cat could also have the ability to speak. But maybe they'll print meow to the screen. A rabbit won't even have the ability to speak at all.

A person might have a list of friends, whereas a cat won't, a rabbit won't things, like that. So we can either add more information. Like, list of friends was an example of that. We can add more behavior. Like, the ability to speak is an example of that. And an example of overriding behavior, like I mentioned, is let's say we have a subclass student of person. If a person's speak method said to print Hello to the screen, we can override that behavior through a speak method inside student, where you don't just print hello to the screen. You can print, I have homework.

So let's try to start implementing this relationship. This is just our animal class. There's nothing new here. I'm just doing a little refresher on what this class looks like. So we've got our init, where we initialize an age and a name that's None. We've got two getters, two setters, and this str method that prints animal, colon, name, colon, age. So yeah, OK, this animal class inherits from object, so the generic Python object.

And now, let's work on the subclass, cat. So when I create my subclass cat, the way I tell Python that this cat is an animal is by putting in the parentheses here the name of the type that I want this class to inherit from. So a cat is an animal. Now, one of the things I kept coming back to is, any time you create a new data type, you have to have an init method. This doesn't specifically have an init method, right? I've just got two other methods here.

So you might think that it's missing. But it's actually not because as soon as you put another data type here in the parentheses, so that cat is an animal, think of it like Python going into the animal class, copying and pasting everything that's part of the animal class or copying everything that's part of that animal class and pasting it inside cat. So since I don't have an init method specifically defined in cat, Python will say, oh, we'll just use the init method of your parent animal.

So the way we create a cat is going to be exactly the same way we create an animal, except that the name is going to be cat as my object type instead of animal. But we just pass it in one thing, which is the age of this cat. So since we're copying and pasting everything-- yeah, question.

AUDIENCE: The parent class of animal [INAUDIBLE] is that also [INAUDIBLE]?

ANA BELL: Yes, exactly. So the parent class of animal is object. So cat will also be a Python object. But that's super generic stuff, like binding a variable name to this object, things like that. So not only does the init get copied in, but every single data attribute, age and name, every single way that that data attribute gets created, so the self.age is going to be a data attribute of cat. And it's going to be set to whatever is passed in as a parameter. self.name will be initialized to none, just like for animal.

I've got my two getters, my two setters that also work with cats. And then, the str method of animal will also be inherited in here. But now, we notice one thing. And that's, we have an str method defined in the animal class. But then, in my cat class, I define an str method as well, right? So that's called overriding your parents's class. And when we create an object of type cat, if this object has a method that has the same name as their parent, we use this method. There's no reason to go up to your parent to ask for their method. We use the one that is for this object.

And cat, in addition to having everything that animal has, implements a new behavior, which is the ability to speak. And all it does is print meow to the screen. OK, so let's look at some code. So here's my cat. So I created a new cat object the same way I would create an animal. But I'm invoking the name of this class, cat. The way I create an animal is just by passing in the age of this thing, right?

So here, I'm creating a cat whose age is five. The name of this cat is None, right? Because that's what the init method of animal does. But I can run the methods on animal on my cat object because a cat is an animal. So all the methods that work with animals will work with an object of type cat. So, here, I can just run the set_name method on my cat object. Even though the method is not explicitly defined in here, it's defined in my parent.

So if I set the name to fluffy and then I print the cat object, it's going to print, it's a cat, colon, the name, colon, the age. speak is just going to print meow to the screen. We can do the getter methods as well. So all of these methods that were implemented with animal work with cats as well. Now, in here, object a was created up here when we talked about animals. It's an animal object because it was created using the animal invocation here.

Does the animal class have a method to speak? No. So if I actually run this, it'll give me an error, right? It just says there's no attribute speak, which makes sense. I never defined that. I defined that in your child, not the parent. Questions about cats? OK, so I want to briefly touch upon overriding methods because it can get a little bit confusing. So you notice, the str method was implemented in both of these objects.

The str method is in cat, which overrides the animal's method to print cat, colon, name, colon, age. And the animal method, str method, prints animal, colon, name, colon, age. So the rule is, when you're running a method that you know exists in a whole bunch of these inherited objects, you look at-- which one is it? It's str, right? So it would be the print method, or any method. It doesn't matter what it is. You look at the object you're calling the method on, right?

So if it's a dot notation, you look at the thing before the dot. If it's one of these special methods, what's the object you're running this method on? So here, I've got the print method on object c. Python asks, what is your type? Oh, you're a cat? Do you have an str method defined? Yes, you do. So then, it uses the one that it finds right away. But if for some reason, the current object doesn't have that method, so an example of that is set_name, right?

set_name is not a method defined in cat. c is an object of type cat. It doesn't have that method. Python says, oh, you don't have that method. Let me look at your parent. Does your parent have that method? And then it looks through in here. And it finds it. Good, if it finds it, it uses that one. If it doesn't find it, it looks at your parent's parent. If your parent's parent has it, it uses that one. And if it doesn't, it looks at your parent's parent's parent.

Until it gets to the generic Python object, this one right here, if they have it, it uses that one. And if it doesn't, then it throws an error. So an example of something that the generic Python object has is the str method, right? It just prints the memory location. And that's why, when we don't implement our str method in our class, Python defaults to the generic Python object. Questions?

OK, let's look at a person. So let's create a person object. This person object, again, will inherit from animal because the only things we set an animal as defined as is being alive for some period of time. And it has no name. The name is None. So we don't even pass that in. So let's say the parent class, the person is animal. But this is my design choice, also to highlight a bunch of stuff. But let's say that this person class, when I create a new person object, I would like to pass in an age and a name.

So I don't just want to create a person with an age. I want to actually create it using a name in that parameter list. So as an example, in my code here, when I create a person, I would like to pass in their name, comma, and the age, two parameters to make a person. Well, I can't use the animal's init method, right? I could for cat because cat was happy to just be created using an age. But I can't do that for a person because I would like to create a person by passing in two parameters in the creation of the person.

So what I have to do is effectively override the init method of animal by implementing it in my class definition, right? So here, I have to define my own init method. And I do it because now, I'm not just passing in an age. I want to pass in a name and an age in the parameter list. And then, beyond that, what do I do inside the init method? Well, I know that this person is an animal. So what I'm going to do to make my life simpler is to call animals init method.

So here, we use this dot notation on the name of the class, sort of similar to how I showed you that long way of calling methods. Well, here's the name of the class, dot the name of the method, init. And now, I pass in all the parameters, self and age. So I'm going to call animal's init method, which will create that self.age, set it to age, and create that self.name and set it to none. So I'm taking advantage of the fact that that init method already does those two lines for me, right?

So I've turned those two lines into one line here. And then I'm going to say, well, I'd like to set the name of my person. So I'm going to call the method set_name with the parameter that's passed in. And then I'm also going to initialize another data attribute for a person, which is a list of friends, initially empty. So what's nice about this, and when we implement the student class, it'll look even nicer, what's nice about this is we're taking advantage of the fact that the init method of animal already does some work for us.

But at the same time, we can clearly see, in this subclass, what the person object brings in addition to the animal object, right? So in addition to just being an animal, we give a name and get a list of friends, right? So it's very nice to see the extra data attributes or what you need to change with respect to the animal to make a person. And then, beyond that-- so I think that's what I said. Sorry, I didn't go through that as I said it.

And then beyond that, I've got some-- we can add some getters and setters. I just did a select few. But you should add them for all of them. So the get_friends just returns a copy of my list because maybe I want to keep my original order or something like that. So it's just good style to return a copy of a list. The ability to add a friend to my list basically just adds a friend's name as a string, if it's not already in the list. So I can't have two Ana's in my list. I consider them the same.

Ability to speak just prints hello to the screen. And then I added this cute little function to tell me the age difference between this object that I'm calling age difference on and some other person, right? And all it does is grab the two ages, take the absolute value of the difference, and print that to the screen. And then lastly, we're going to override the str method of animal, instead of saying animal, colon, name, colon, age, to say person, colon, name, colon, age. So this way, it helps me figure out the type as well.

So in my code here, I've got two people, p1, p2. Here's Jack, age 30. Here's Jill, age 25. If I run the get_name, get_age on both of these, this will run animal's get_age get_name. I've not defined these in here, which is fine. We inherit from animal. And animal knows how to grab the age and name. So there they are.

If I print P1, it'll print person, colon, name, age. If I ask p1 to speak, it just prints, hello. If I ask the age difference between p1 and p2, no matter what, it just takes the absolute value, prints five year difference. And then let's add some friends to p1. So here, I've got two Bob's. But it's just the list, keeping unique names.

OK, so let's have you try this for a little bit. It's a little bit, again, working with objects of this type. So it's a function that takes in a dictionary. So I'll tell you what the dictionary looks like. It maps a person object to a cat object, all right? So that's my dictionary. So this is the key. This is the value. So I've got all these person objects being mapped to cat objects.

So as an example, here's an input dictionary. p1 is this person here. And p2 is this person here, right? So my two keys, p1, p2, are person objects, right? They're not integers, floats, strings. They're person objects. And then the values associated with those are cat objects. So here's an object of type cat with its name. I just ran set_name on that cat after I created it, same here. Here's the name set to this new cat object.

So I've mapped p1 to c1, p2 to c2. So if I run this function, what I'd like to do is not return anything this function. It just prints something. On each line, as you're going through all the items in the dictionary, it just prints the name of that key, colon, the name of the value. So all I'd like to do is write code that extracts the name from my person object and from the cat object, OK?

I know what you're thinking. I look really young for 86. But it's diet, exercise, and hanging out with you guys, and candy, for sure. So here, let's write this code on 178. All right, does anyone have a start? Yeah. d.items, yep. Let's write a note for ourselves. k is person. v is cat, yep.

AUDIENCE: [INAUDIBLE] k.get_name.

ANA BELL: Yep, so k.get_name, you want to save it as a variable?

AUDIENCE: [INAUDIBLE]

ANA BELL: Or no?

AUDIENCE: [INAUDIBLE]

ANA BELL: Oh, you want to put it on the one line. That's fine, yep. Print, k.get_name.

AUDIENCE: [INAUDIBLE]

ANA BELL: Yup, v.get_name, exactly, yep, perfect. And yeah, nothing to return. So let's run that, cool. Does anyone have questions about that? All right, so we're just manipulating these object types. And again, if it's confusing, I highly recommend, quiz situations and things like that, now that we're working with object types, just make little notes, right? I know we're iterating through a dictionary. And it's kind of convention, right? Keys are integers, things like that.

But this particular case, just a little note that k is a person will help you remember that you need to run a method on this k variable, like we did here, k.get_name and then v.get_name. OK, yeah.

AUDIENCE: How do we ensure that the get_name, that the [INAUDIBLE]? And--

ANA BELL: How do you ensure that the keys are person? You can't ensure it in this particular case. I mean, you could say, if type of k equal, equal, person, capital P, person, then do the code, and else probably just skip it or raise a value error or something. Like, you could enforce it that way. But in this particular case, we're just assuming that the tester will make person objects mapped to cat objects, yeah. But yes, certainly, if you're making a software for something more complex, you should probably make sure of that, enforce that.

OK, so the big idea with inheritance is that now that we have sub-classes, also known as child classes, those sub-classes use a parent's attributes. So everything that a parent has and can do, a child has and can do as well. But that child can override certain parent's behaviors. And the child can add new behaviors or new attributes, in addition to the parent. Let's look at one more subclass student before we go on to one last thing.

So a student here from our pictures and diagrams inherits from person, not from animal, but indirectly from animal, right? So a student is a person. And when I create a person, I would love to create it using a name an age and a major. But we can use a default parameter for that major to be None if we don't actually want to pass it in. But I would like to create it by setting their major as well.

So now, I can't use the parent's init method because I've got three parameters I would like to initiate my student with. So I would like to create my own init method inside person. So here I am, defining my own init method. And now, it becomes apparent why it's nice to call the init method of your parent because if I say a student is a person, all I need to do to initialize a person type, all the attributes associated with a person and the init method of the person. So just call the init method of the person.

That will create my name, my age, set my name, create my list of friends, all that stuff. So those five lines get compacted into this one line. And then it also becomes really easy to see what the student has, in addition to the person. Well, it just has a major data attribute. self.major is set to whatever is passed in. And then, beyond this, it's just methods here and there to do stuff. So here, I've got a change_major method.

It just sets the major to something. I should probably add a getter in there as well. But I ran out of room. And here's a speak method that gets overridden from the method of person. So the speak method, for student, I made it slightly more complex than what the parent has. So here, I'm using this random library, not a random library I found, arbitrary library, it's a library called random. And it has a bunch of functions that allow you to deal with random numbers.

So one of the functions that this random library has gives you a number between 0 and 1 at random, so a float at random. So what I'm doing in the speak method for students is randomly printing one of four strings, according to where that random number that's gotten lines between 0 and 1. And then-- oops, not yet. And then, here, I'm overriding my str method.

So we can see, in the student class here, here, I've created two students. So this one actually has a major. This one's major is going to be set to None, just the default value. And then, if I run this code, you can see, every time I run it, the student 1 says something different. Student two says something different. So it's just running this random number and then choosing what to print. Maybe more often than not, I should bias it towards something.

All right, so one more class I'd like to talk about, rabbit. That's the one that we actually haven't talked about from those little subcategories. And as we talk about this rabbit class, I'd like to introduce one more idea of a variable. So far, we've had just plain old variables that go away as soon as a environment disappears. We've talked about instance variables, a.k.a. data attributes, which are consistent for objects that you create of a certain type but have different values for different instances.

The last variable I'd like to talk about is a class variable. What's cool about a class variable is that, think of it like a shared resource. So it's a variable that any instance of this particular type can access and modify. And if it's modified, all the other instances will see this modified value, right? So it's just shared across all the instances of type rabbit, in this particular case. And so there's many different ways to use class variables. In object-oriented programming, they're pretty useful.

The way I'm going to use it here is to give me the ability to basically count how many instances of this type rabbit I've created in my program. So when I run the program, remember, I can create a whole bunch of instances. I'm going to try to use this class variable as a way for me to basically keep a counter of how many of these instances I've created. All right, so let's look at the code.

So the first thing I'm going to do is just inherit from animal. It gets a name and an age. And that's about it, all those getters and setters and the str method. Now, to create my class variable, notice I'm defining this variable, just plain old variable, outside of any methods within the class definition, right? So here's tag is equal to 1. The very first variable-- the very first instance of a rabbit I create will grab the value of whatever it says here.

But then, if any instance changes this value, other instances will see that changed value, OK? So what we're going to do is, we're going to implement ID numbers for these rabbits, so sort of tagging them to keep track of how many there are. So in the init method of animal, or of rabbit, I'm going to create a new rabbit using an age and two parents. So, again, different than animal, so I'm going to have to implement my own init method.

But I'll call animal's init method because it does some work for me. Then, I'm going to add two data attributes for the two parents to be whatever is passed in. And then, down here is where I'm going to use this class variable, the shared resource, these two lines. So the first thing I'm going to do is add one last data attribute for my rabbit, which is the rid value. So it's the rabbit ID. And this is going to be a unique value for every rabbit I create.

First rabbit will have a value of 1 that I create in my program. Second rabbit I create will have a value of 2, and so on. So what am I setting it to? Well, I'm going to set it to whatever the tag is. So the very first rabbit I create, their rid will be 1. That's what the tag is initially set to. But then, before I finish the init method, there's one other line of code, rabbit.tag plus equals 1. So this instance, right before it finishes creating itself, is going to take that tag and increment it by 1.

So the next rabbit I create, it's going to grab the tag value that was just changed, OK? Let's visualize it. So we're going to do it with actual rabbits. OK, so first, I'm going to-- so there's going to be three lines of code. And this is the program I'm going to run. So the first thing I'm going to do is create my first rabbit, right? Its rid will be whatever the value of tag is originally, right? So, originally, we said the tag is 1.

So behind the scenes, what's going to happen is Python says, oh, you're the first instance of rabbit class. So the tag was initialized to 1. So your rid is going to be whatever the value is, 1. So I've got, this rabbit, its age is 8. Two parents are None. And rid is 1. But then, before I finish creating this rabbit, the last line of the init method says, take the tag and increment it by 1.

All right, next line in the code says, here, let me create another rabbit. This one, I'm going to pass in age 6 as my parameter. So that's the age 6. Two parents are None by default. So Python says, all right, well, here's a new rabbit object. It's age is 6. The two parents are None. A line that says self.rid, so the rid of r2 will be whatever tag is right now. Well, the previous rabbit incremented it to 2. So the rid of this next rabbit is 2, right?

OK, the last line of code before this rabbit finishes it creating itself is to increment the tag to 3. So now, if I have one more line of code, I'm creating one more rabbit. This age is 10, right? So behind the scenes, Python creates this variable named R3. It's bound to an object, a rabbit object, whose age is 10. Two parents are None, of course, because we didn't pass in any parents. And the rid is whatever the tag is right now, 3, OK?

Well, here's the one with rid of 3. And before we finish creating, let's just increment the tag so that we set it up for the next rabbit, OK? Everyone OK so far?

AUDIENCE: [INAUDIBLE] just create the first line [INAUDIBLE]?

ANA BELL: Yep, yes, yes, it gives you two because when you run this line, rabbit 8, it has to run the init to completion. And the last line of the init always increments it to be one more than what it started with. You can't, I guess, pause the function run in the middle to check. OK, so let's look at a couple other methods that we can implement-- sorry, other questions about that very cool way of creating rabbits? Yeah.

AUDIENCE: I guess I just wanted to know more about the space above the definitions, so the [INAUDIBLE].

ANA BELL: Yes, let's go back, here.

AUDIENCE: Yeah, so above the init definition, that space there, what else-- like, what goes on there?

ANA BELL: Mostly just this. Yeah, mostly, you want the object to have things associated with it. So, really, shared stuff is nice. But it's a little tenuous in using it just because-- you should use it for pretty specific situations, right? You don't just want to define a whole bunch of variables that everybody can access here and there, only specific situations. Yeah, most of the time, you just have methods in the definition, yeah. But maybe there's other stuff. I just don't know about it right now, yeah.

OK, let's look at a couple more methods for the rabbit so here, I've got a getter, just three getters. I should probably put-- I don't want to put a setter for the rid because that would mess up my counting. And probably, I don't want setters for parents, too. But maybe we might. I don't know. The only thing that looks a little bit weird for the get_rid is this zfill.

And I added that as a cute little thing to basically make the ID look like an ID number. So it pre-fills the front with zeros. Like, it pads the front with zeros. So for the ID of 1, you can see, it's 00001. For an ID of 123, it would be 00123, right? So it just makes it look nice when we print it out, when we print out the ID. And otherwise, the two parents just return the parent objects.

One interesting method that I would like to add, and we'll play on the fact that rabbits mate here, is to add two rabbits together. So we're implementing the dunder method, double underscore, add, double underscore, to have the ability to add two rabbits together in our code, all right? So, again, this is a design decision I made. So when I add two rabbits together, I'm going to create a new rabbit object. And that's exactly what the code is doing inside here.

So I'm going to run this add dunder method on self and other, right? And then, behind, or in front of the scenes, I guess, is going to be this plus operator. So the self will be the thing before the plus. And the other will be the thing after the plus, right? Just like what we saw last lecture. So when we add r1 plus r2, what I would like the result to be is another rabbit object, who has one parent, r1, and the other parent, r2. Those are the things we added together. And let's say this new rabbit object is age of 0, right?

It's a newborn. So to implement that, we just have-- we're returning a new rabbit object here, right? So we're just creating a new rabbit object on the fly in this method. How do we create a rabbit object? We need to give it an age and the two parents. Originally, when we created those three r1, r2, r3's, they didn't have parents, right? They were just unknown or something like that. But in this particular case, we do want to know what their parents are. Their parents are the thing before the plus and the thing after the plus.

So one parent will be self. And the other parent will be other, the thing that's in the parameter list. So let's continue on with our program here. We had these three lines of code that were run. And I created these three rabbits with these IDs, 1, 2, 3. If I add two rabbits together, r1 plus r2, to give me a rabbit object variable, e4, Python says, all right, well, let me run this dunder method behind the scenes of the plus.

So r4 effectively becomes what? Well, we replace, in the previous slide right here, the return is rabbit(0), one parent, comma, the other parent. So when we make this addition, we have rabbit(0), comma, one parent, the thing before the dot, comma, the other parent, or the thing before the plus and then the thing after the plus. So my r4 becomes the result of adding r1 plus r2. So its parents are these two.

, Now how does this rabbit get created, right? It's a new rabbit object. So we run the init method of the rabid object, which means that, here's a variable. It's bound to a rabbit object. Its age is 0. It has these two parents that our object is bound to other rabbit objects, up here, r1 and r2. And the ID, just like before, is whatever the tag is right now. Well, we already created three rabbit objects ahead of this one. So this one's tag will be 4.

And then, right before we finish, we increment the tag to 5. So no matter how we're creating these rabbit objects, either just plain old in our program directly or through an indirect method, in this case, the plus, we're still creating rabbit objects in our program, right? So that counter, that shared variable tag, is still coming into play, right? So we're still counting all of these rabbit objects created.

Does that make sense? OK, good. So, yeah, that's fine. So one last method, so this is a method that checks for equality between two rabbits. And again, my design choice is to say that two rabbits are equal. So if I say r1 equal equal r2, that will tell me true or false. And my design choice is to say that two rabbits are equal if they have the same parents. So if I create another rabbit object, 4 was r1 plus r2.

But if 5 is r2 plus r1, I want to say that 5 and 4 are equal because they have the same parents, right? I don't care that it was r1 plus r2 or r2 plus r1. They have the same parents. It's just an opposite order. And so that's what this eq method is doing. It's a dunder method to implement equality between two rabbits. So parents_same is a Boolean here that just checks the rid. So this Boolean, parents_same, is going to check that the addition was made, r1 plus r2, r1 plus r2, right?

And parents_opposite is also going to be a Boolean, either true or false, that checks if I made the rabbits, r1 plus r2, and then r2 plus r1, so backward in the parents. But they still have the same parents. And the reason I'm checking for IDs is because IDs are unique. So originally, when I wrote this code a long time ago, I actually ended up, my first iteration, checking just the straight up parents values, right? So it was comparing, basically, rabbit objects together.

But the problem with that code is that, at some point, it tried to compare a none, some rabbits might have a none as their parent, with an actual rabbit object. And then the code crashed. And then I realized, I can just compare the ID values directly because those are, one, just numbers, so very easy to compare. And two, they're unique. So I know I'm not going to have two rabbits with the same ID. And so, in this particular case, I've got these two rabbits. I should say they're equal. But then, if I add r2 plus r3, r6, this one is not going to be equal to any of my other rabbits.

So here's my code. So here, I've got my three rabbits, right? So I think we've printed this out already. So here's r1. It's a rabbit with this ID, rabbit with this ID, rabbit with this ID. And then r1's parents, r2's parents, and r3's parents all have none, are none. But then, when I add r4 as r1 plus r2, I can print-- r4 is a rabbit with ID of 4. And then R1 and R2 are, as usual, what we just saw.

And then, when we grab the parents of r4, it's going to be r1, which is this rabbit with this ID, and r2 with this rabbit with this ID, right? And then, we can check the equality. So here, I can create r4, r5, and r6. So r3 plus r4 and r4 plus r3, they should be equivalent, right? So here, I've got r5 and r6 down here. See, I'm just running the double equal sign on objects of type rabbit, which is pretty cool.

And they're the same, right? Because they have the same two parents. I don't care that they're in opposite order. But then, r4 and r6 have different parents, right? r4 had 1 and 2. And R6 had 3 and 4. Questions about this code? OK, so class variable is pretty cool. You share them across all the instances. So if one instance modifies it, they'll be modified for all the other instances.

So we have one more example to look at next lecture. We're actually going to implement our own fitness tracker class. So it's going to be a little bit more complex. But we're going to see a lot of the same ideas that we saw today, just in this slightly more complex setting of implementing our own fitness tracker. So it's still kind of an abstract thing, but more useful than animals and rabbits and person and student classes.