KENDRA PUGH: Hi. I'd like to talk to you today about inheritance as a fundamental concept in object oriented programming, its use in Python, and also tips and tricks for using inheritance in Python and in the object oriented programming paradigm in 6.01. First thing I'm going to do is give a really quick crash course on inheritance to catch you up to speed, and also so that you're clear on what I mean when I say something like parent class. And also, I'm going to address the sort of nuances of using inheritance while programming in an object oriented fashion in Python.

A lot of the code in 6.01 uses the object oriented programming paradigm. And a lot of the code in 6.01 will inherit from parent classes. So this is part of the motivation for going through a quick review, and then also indicating the most common slip ups, and also the most significant things that you may or may not have seen from other languages.

All right, first, a crash course on inheritance. Let's look at the board. Inheritance is the idea that you can arrange objects in a hierarchical fashion such that very, very generalized or basic things that are true of a whole group of objects can be specified at a higher level. And then, you can work your way down to progressively more specific levels.

The most formal encounter you've probably had with this thing, that approach, is the biological taxonomy, right? Kingdom, phylum, class, order, family, genus, species. Every species has all the properties of that particular genus. All the genuses of a particular family have the properties of that family, and so on, and so forth. That's a very concrete example. But I find it a little boring. So I'm going to talk about dog breeds instead.

You're probably familiar with the fact that golden retrievers are a type of retrievers and that retrievers are a particular kind of dog. You can make generalizations about goldens based on what you know about dogs, right? All dogs bark. All dogs have four legs if they aren't injured or have some sort of congenital defect, that kind of thing. And goldens also have all the properties of retrievers, right? They are capable of going and catching game that you've either shot, or possibly chase it down and bring it back to you. So they're bred to have very particular properties.

Goldens are also bred to have very particular properties. And those that are very specific to goldens define the difference between a golden versus a retriever in the general sense. Likewise, when we want to make objects that have very particular properties but also share general properties with other objects, we're going to create a new category of object and put the specifics in that very specific category and then take the things that we can generalize and put them in more general categories so we don't end up rewriting a lot of code. Or we end up reusing code, but not copying and pasting it everywhere because that's annoying.

The other major advantage of using inheritance is that code is more intuitive. You can make references to the same piece of code all over the place. But it's not as intuitively accessible to do that over, and over, and over again, right? It's really convenient to think of the fact that golden could be a subclass or subtype of retriever, and that retriever could be a subclass or subtype of dog.

When I talk about this relationship in terms of object oriented programming-- when I talk about these categories in terms of object oriented programming and when you're actually looking at code, goldens are a subclass, or child class, of retrievers. And retrievers are a parent class or super class of goldens. Likewise, dogs are a parent class of retrievers.

So now, I've defined my terminology and also hopefully given you a very, very, very quick review of inheritance. Now, I'm going to talk about the specifics in Python.

If I turn over here, I've written up a very short class definition for dog, right? Every dog has the class attribute, cry. Every dog has an initialization method that gives every dog a very specific name that is passed in when you initialize the dog. And every dog has access to the class method, greeting, which returns a string that says, "I'm," whatever the name of the dog is, and also the specific cry, which in this case, is actually the class cry.

If you're unfamiliar with using the plus in terms of strings, it's just a concatenator. So play around with that in IDLE if you're confused. I would recommend copying all of this into IDLE, and then playing around with a particular instantiation of dogs, in this case, Lassie.

If you look at Lassie.name, you'll end up going after self.name, which is specified when your initialize the object. So Lassie's name is Lassie.

Likewise, if you were to type in Lassie.greeting, open paren, close paren, and hit Enter, you should get a string return that says, "I'm Lassie," comma, "bark." Mostly this is to familiarize you with object oriented in Python in the general sense. Now, we're going to look at what happens when you want to set up a subclass.

If I set up class Retriever and I want to inherit from the super-class, Dog, I'm going to pass in Dog. This is in the same syntax that I would use if it were a function and I wanted to pass in a parameter. If I wanted to inherit from multiple things or multiple classes, I would put multiple classes here. Right now, we're just going to inherit from Dog.

Note that I have no code here. This is pretty much meant to explicitly specify the fact that Retriever is not actually going to introduce any new properties to dogs. Their types are going to be different. So if I create something that's a Retriever, it will be of object type Retriever, versus if I create something and say, "Dog," open paren, close paren, it's going to be of type Dog.

But what happens when I create a Retriever-- and as an aside, if you know who Benji is, I know he's not a retriever. But bear with me here. If I create a Retriever, it's first going to look for an initialization in any other methods or attributes in the Retriever class definition, run any code that's here, and then go to the parent class, and run all the code here.

So even though Retriever did not have any explicit code underneath it, I can still interact with the object, Benji, the same way that I interacted with the object Lassie. It has all the same methods and all the same attributes. Phew. So there's basic inheritance.

And I will make another aside that if you're doing this, you probably don't need to create a subclass in the first place. If you're designing your own code, and you're trying to think about what the best way to organize things is, if you have to create a subtype or a subclass and there are no new methods or attributes or no different ways of addressing those methods or attributes, then this category is probably actually just this category. You may want to make a difference so that you can do interesting things with type checking. I think that's the only thing I can think of that would justify it. And I might be wrong. Python gurus out there should correct me. But a thing to keep in mind.

So we've done the first half of our inheritance. We're going to inherit one more time and create a class of golden retrievers. Once again, I've got my class definition and my indication that I'm going to inherit from Retriever. I don't have any initialization or attribute assignments. I only have a definition for greeting.

So what happens here? Well, the first thing we always do is look for an initialization method. Golden doesn't have one, so it's going to check the Retriever class. Retriever doesn't have one, so it's going to check the Dog class. The initialization method is here. So when it runs the initialization method, it's going to run this code.

The first thing that's going to happen is any code, or any attribute assignments, or method definitions here are going to be considered the canon, or the first thing that any Golden is going to reference. So greeting is going to be executed before greeting used in any other place. You notice the only difference between this greeting and the Dog greeting is that "OHAI!" has been prepended to the phrase.

And the way that we end up doing that is we refer to-- we concatenate, and then refer to the superclass. And once again, we have to pass in the explicit argument, self, when we're talking about a class definition. Later, when you actually instantiate an object and use your parens, you're not going to have to put self as an argument. It'll get confused. We'll go over that in a second.

So let's say I create a golden retriever, Sidney. I'm going to pass in one argument, which is the name. We're going to consider all the definitions here first, which means that goldens are going to have a method for greeting that is specified here. It's going to use the method for greeting from Retriever. And we could put in anything here, right? We could put Dog.greeting. We could put in some other function that is in the same environment as class Golden. But here, we can explicitly access the superclass that we defined here.

We're going to head over to Retriever to see if there any additional methods or attributes that are a consequence of being a subclass of Retriever that we need to add to our definition. Now, we just hit the pass. On the other hand, Retriever inherits from Dog. So once again, we have to jump over to a super-class and grab any attributes or methods that are defined there as well.

So all the way back over to Sidney. When I call Sidney.greeting(), the first thing that happens is that I look in the most specific subclass, or whatever my object type is and see if there's a definition for the method. Because there is, I'm not going to use Dog.greeting(). I'm going to use Golden.greeting(). Golden.greeting() says return a string that says, "OHAI!" And also append it to whatever Retriever.greeting() returns.

I go over to Retriever. It's not here, but I still have a reference to Dog. I go over to Dog. It has a method for greeting. And it says, "I'm Sidney. Bark." So the final return type should be, "OHAI. I'm Sidney. Bark."

This concludes my basic overview of inheritance of object-oriented programming in Python for 6.01. Next time, I'll review some interesting features in Python that actually originated in earlier languages and also particularly things in aliasing that people that are new to Python or people that are new to programming find especially confusing.

MIT OpenCourseWare
http://ocw.mit.edu

6.01SC Introduction to Electrical Engineering and Computer Science
Spring 2011

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.