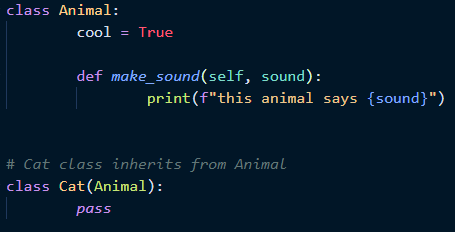
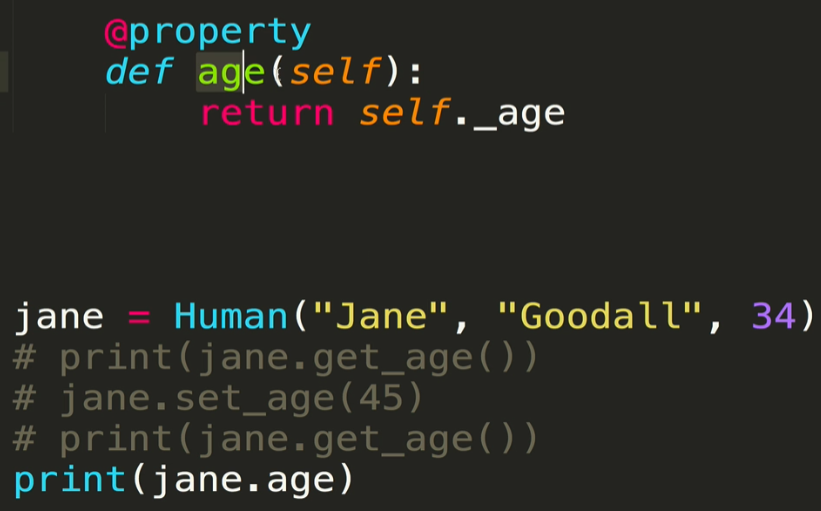
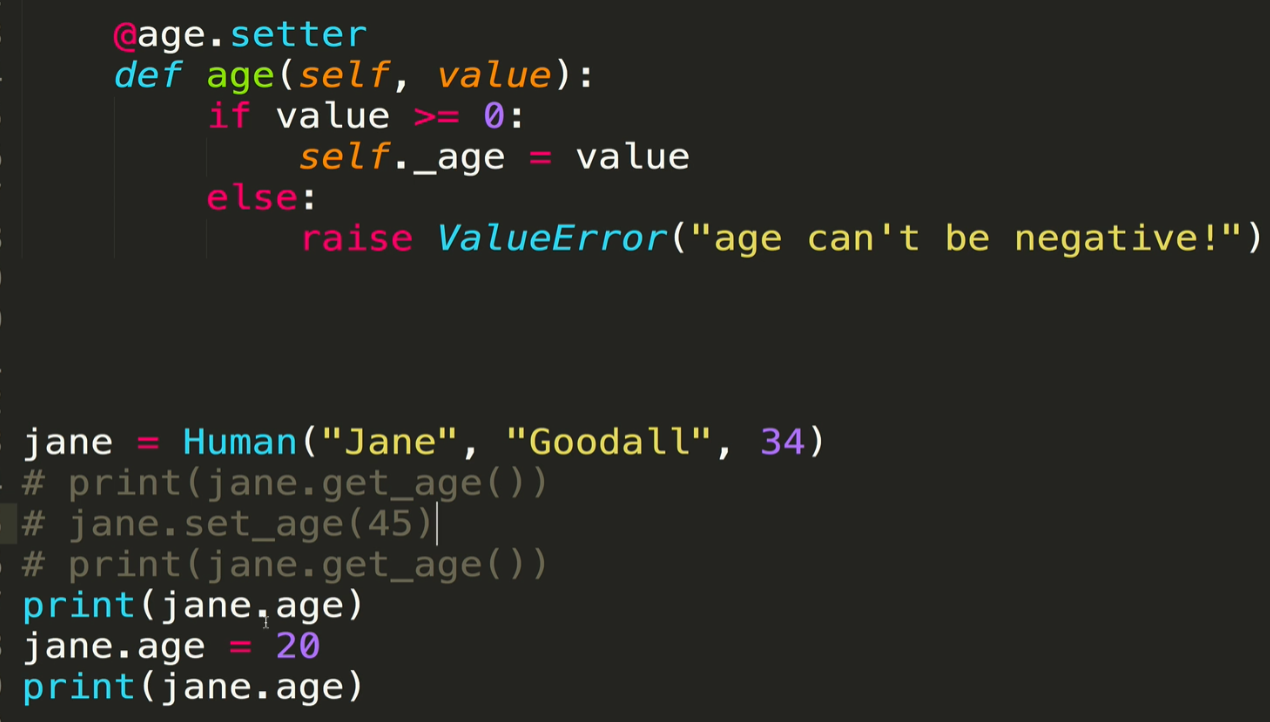
* **Inheritance** is a fundamental aspect of object-oriented programming
  + The idea behind inheritance is that similar content can be shared between different classes
    - Example: User, Moderator, and Admin classes (for Reddit for example) can share methods like “login”, “comment”, “logout”. Instead of defining these methods for every class, we simply allow classes to inherit properties from another class
  + In Python vernacular, the “child” class inherits properties from a “base” or “parent” class
  + To create inheritance relationships, we *pass the parent class as an argument to the definition of the child class*
    - In the example below, the Cat() class is created that inherits the properties from the Animal() class



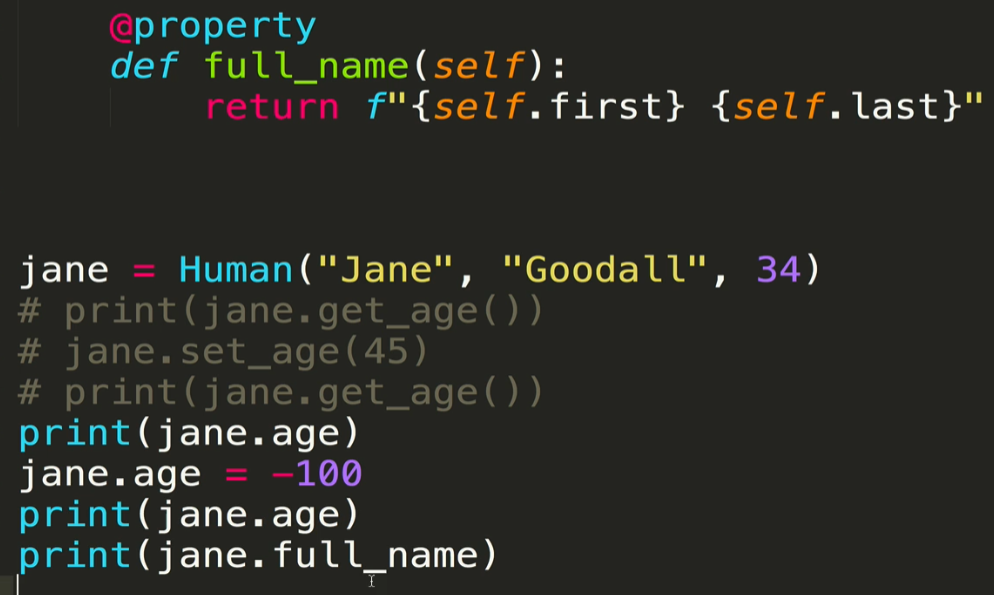
* In OOP, **properties** can be used to restrict what can be done with your objects
  + You start by using the @property decorator. It alters how the following function works
  + Now you can simply call the property without parentheses, and it will return the attribute that you set within the function.
    - In this example, the property age() is a function that returns the value of self.\_age. When calling this property, parentheses are NOT used. And remember that developers are never supposed to call attributes directly that are preceded by an underscore.
      * This is an example of a “getter” function



* + Properties can also be used as “setter” functions. These functions alter “private” properties of the objects
    - In this example, the @age.setter decorator is used. We define another function called age() that takes in a value. This function sets the new value of \_age equal to the value provided. It is likewise called without parentheses and is set equal to the value that is to be passed in



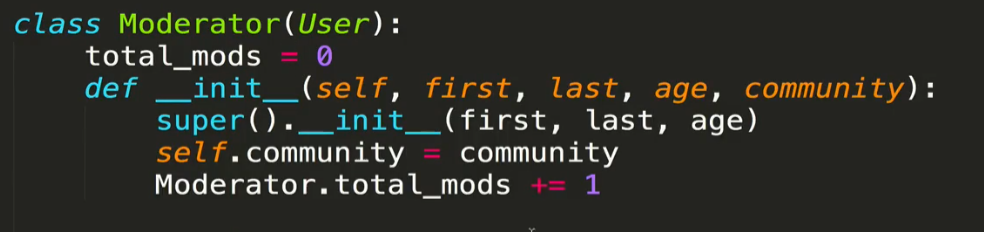
* + One more example of property for full name. here



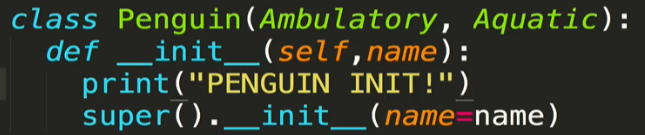
* + Important note: properties are NOT attributes, and therefore will not appear in the list of attributes and methods when running *Object.\_\_dict\_\_*
* In Object inheritance, the **super()** method allows us to avoid duplication of attribute definitions
  + super() allows you to use the \_\_init\_\_ function of the parent object to define attributes of the child object. It always refers to the base or parent class of the current class
  + When using super(), *self* no longer needs to be passed in
    - This is NOT true when directly calling the parent class without using super(). See below.
  + In the example below, the super() function is used within the Cat class to access the \_\_init\_\_ function of the parent class Animal. There, the name and species are defined as attributes without having to be explicitly assigned within the Cat class. It happens behind the scenes with super()!
    - The super() function will work so long as you pass in all of the arguments that it requires



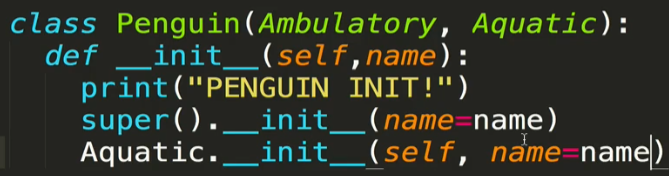
* Just like any other class, a child class that inherits properties from a parent class can have its own unique class attributes that are separate from the parent class attributes
  + In this example, the Moderator class inherits the properties of the User class. However it also has its own class attribute called total\_mods which is independent from the User class.



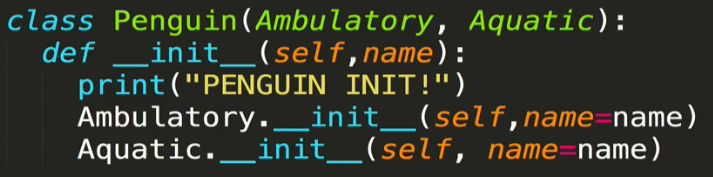
* **Multiple inheritance** is exactly what it sounds like – a child class that inherits properties from *multiple* parent classes
  + Not particularly common
  + So what happens if a method of the same name exists in both parent classes? For example, which one will get called when we use super().\_\_init\_\_? In the example below, the Penguin class inherits from both the Ambulatory and Aquatic classes



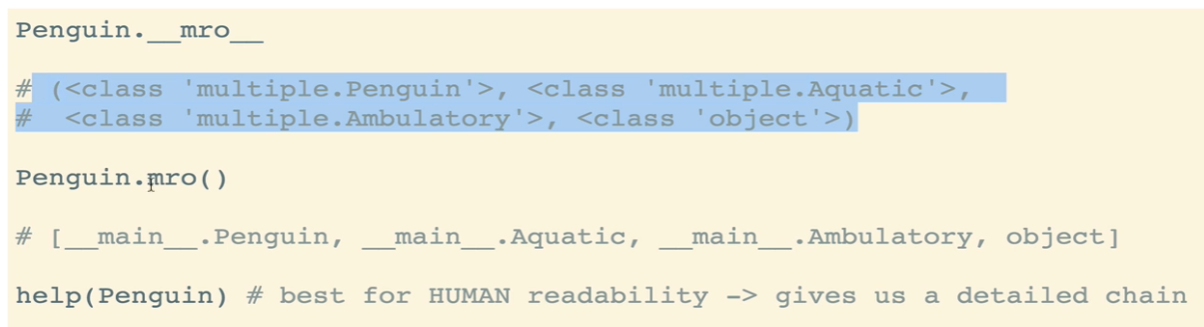
* + - The simple summary is that the first class that is passed into the child class definition is the one whose \_\_init\_\_ function gets called by super()
    - However, we still have access to all methods of both parent classes
  + In order to get both \_\_init\_\_ functions to run, directly call the second parent class. Note that when a parent class is directly called, *self* must be passed in



* + - Colt recommends being explicit and directly calling both classes if you want both \_\_init\_\_ functions to run, as opposed to using super()



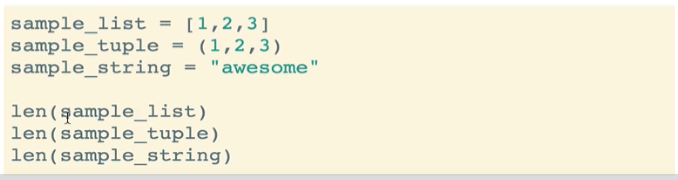
* + Note that this applies for any methods that have the same name in two or more classes that are inherited. It is not limited to just the \_\_init\_\_ function
* **Method resolution order (MRO)** is set whenever you create a class. It defines the order in which Python will look for methods on instances of that class
  + Crucial in situations where, for example, you have methods of the same name in two different parent classes
  + You can programmatically reference the MRO in three ways:
    - \_\_mro\_\_ attribute on the class
    - Use the mro() method on the class
    - Use the built-in help(cls) method
      * Gives you a nice human-readable method resolution order, unlike the other two which return python types or lists



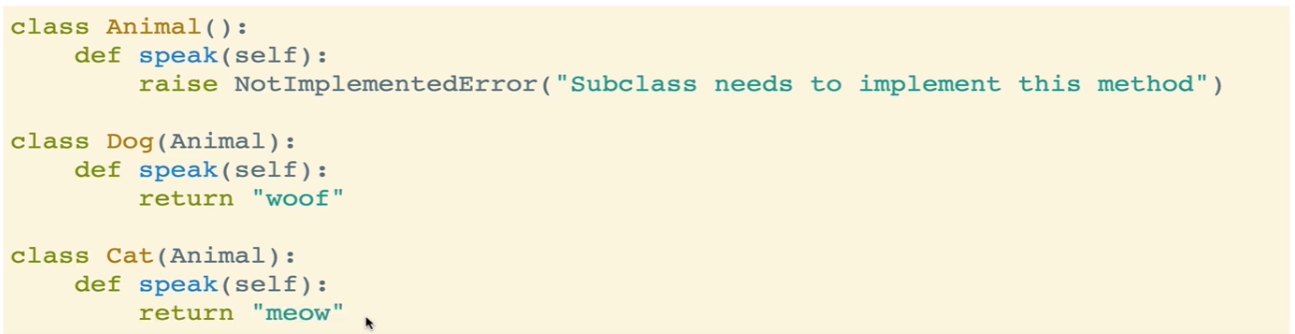
* **Polymorphism** is the idea than object can take on many forms. Two most important practical applications of polymorphism:
  + The same class method works in a similar way for different classes
    - In this example, three different classes all have a speak method that behaves similarly for all of the classes



* + The same operation works for different kinds of objects
    - In this example, the len() method can be used for different kinds of objects



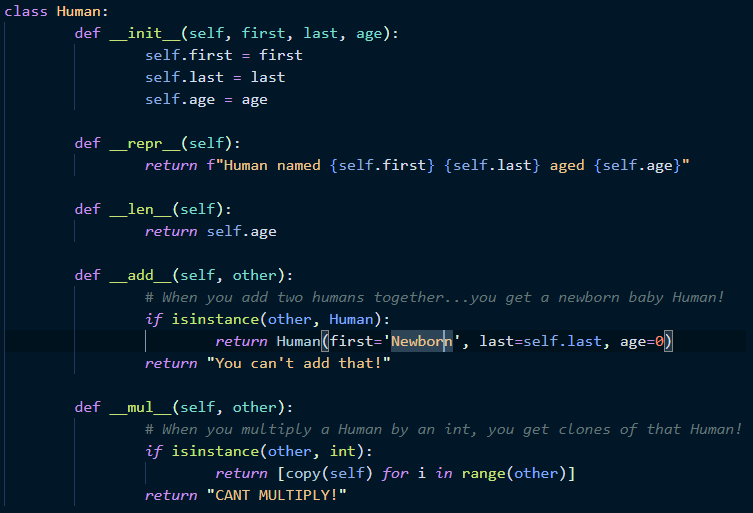
* **Polymorphism & Inheritance**: examples of the two forms:
  1. The same class method works in a similar way for different classes
     + A common implementation of this is to have a method in a base (or parent) class that is overridden by a subclass. This is known as **method overriding**.
       - That is, every subclass will have a different implementation of the method, though the method name will be the same



* 1. The same operation works with different kinds of objects
     + Example: addition of integers vs. addition of strings. This works because the “+” operator is shorthand for a special method called \_\_add\_\_() which works differently depending on the class/object it is being called on
       - The first argument in this operation defines what is done by the + operator. The actual operation that Python performs looks something like x.\_\_add\_\_(y)



* + - In fact, you can define how these special methods work
      * Here, we are defining what happens when len(), add(), and mul() are called on an instance of the class Human
        + For the add() method (which is called using the + operator), a new Human instance is generated. Note that add() is called on the first variable, and will look for that method on the first variable
        + For the mul() method (which is called using the \* operator) will return new copies of that object instance, the number of copies determined by the integer passed in to the function as the second argument. The copies will be put into a list. Note that copy() method from the copy module is needed for new unique objects to be created. If you do not use copy, all of the “copies” will simply be the same reference the same instance



* + Link to Python special methods: <https://docs.python.org/3/reference/datamodel.html>