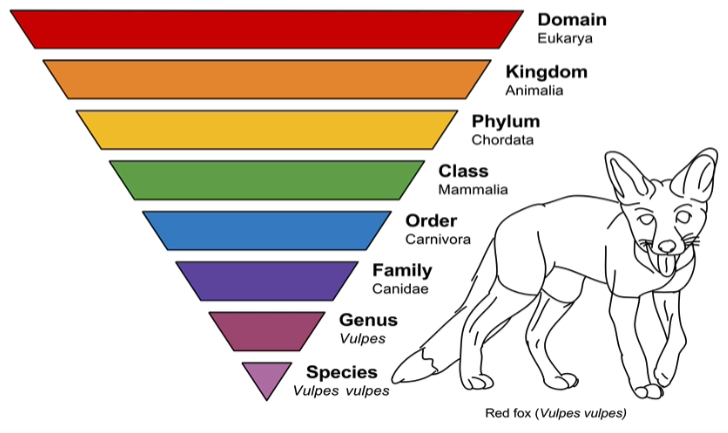
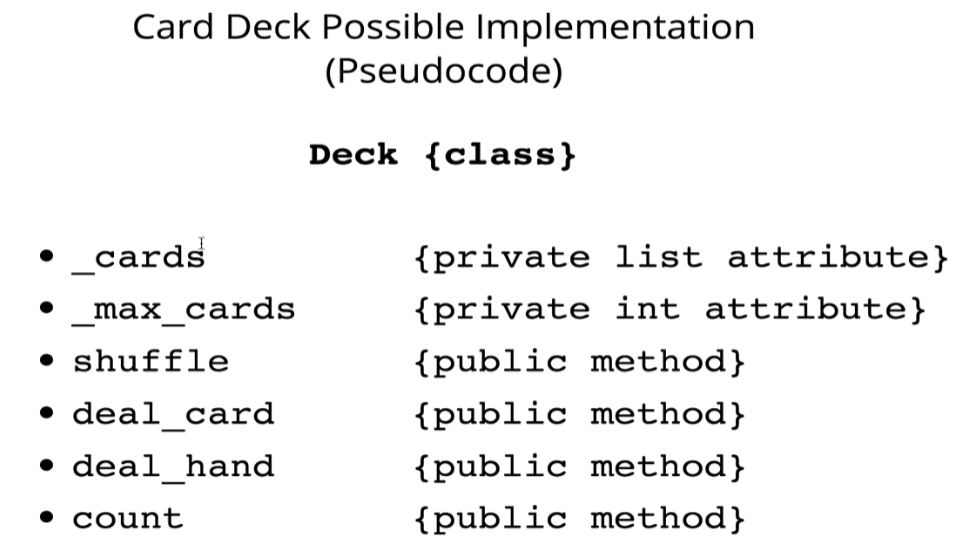
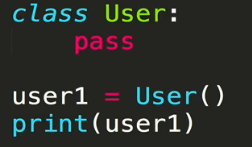
* Object-oriented programming (**OOP**) is not just rote memorization, but rather is a new way of thinking about coding
  + OOP is NOT unique to Python – many popular high-level programming languages are object-oriented
  + OOP is about using code to represent things that exist in the world, things that don’t “exist” in our programming language by default
    - A car on the road
    - A deck of cards in a casino
    - A song
    - A schedule or calendar
  + We do this by using **classes** and **objects**
* A **class** is a blueprint for objects. It defines what every single *instance* of that class should contain.
  + Classes can contain methods (functions) and attributes (similar to keys in a dictionary)
  + Example: A user class can have attributes username, password, and email. It also has methods such as logout(), checkout(), and update\_password()
  + Other examples include the classes we’ve used non-stop in this course, such as int, list, string, etc.
    - However, build-in classes are usually used differently than custom classes, which we typically have to call with the class name
* An **instance** is an **object** that is constructed from a class blueprint
  + The instance contains their class’s methods and attributes
* Why do we use OOP?
  + OOP doesn’t necessarily allow us to do anything that we can’t do without OOP. However, it makes structuring and organizing things in a “human” way much easier
  + With OOP, your goal is the *encapsulate* your code into logical, hierarchical groupings using classes so that you can reason about your code at a higher level
    - Note unlike taxonomical classifications

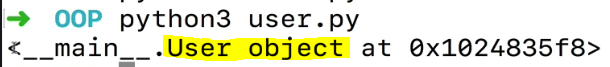


* + It’s just a “nice thing” we can do with our code.
  + Example: In making a poker program, we might have classes for game, player, card, deck, hand, chip, and bet.



* **Private** and **public** entities (basic intro, more to come later)
  + Some things don’t have to be exposed to the programming outside world. For instance, we don’t need to directly access the list of cards. We instead access it indirectly with other methods within that class
  + By contrast, some other methods or variables may need to be exposed outside of the class
  + Python does not actually support true “private” method or variables. So we identify these by putting dunders (double underscores) to let other developers know those variables should only be used from inside the class, not outside
* This leads to **encapsulation**, which is the grouping of public and private attributes and methods into a programmatic class, making **abstraction** possible
  + Example: In design a Deck class for a card game, you can make the *cards* a private attribute
    - If *cards* is private, you can access it using a public method that words with *cards*. For example, we can use a Deck.count() method to count the number (length) of cards
  + **Abstraction** is the idea of exposing only the “relevant” data in a class interface, hiding private attributes and methods (aka the “inner workings”) from users
    - Real-world example: you don’t need to know how steering in a car works mechanically. You just need to know how to turn the wheel, and the car’s mechanisms take care of everything else without you needing to understand it
* Creating a class and an instance
  + The Python keyword for defining a **class** is “class”, followed by the class name and a colon
    - Conventionally, they are named in the singular and camel casing is used
    - Classes must be called
    - Classes must then be called with parentheses
  + To create an **instance**, we call the class and set it equal to a variable name
  + In the example below, we are defining a class called “User” and creating an instance of User called “user1”, then we are calling it to verify that is a User object. Furthermore, the type is of the User class

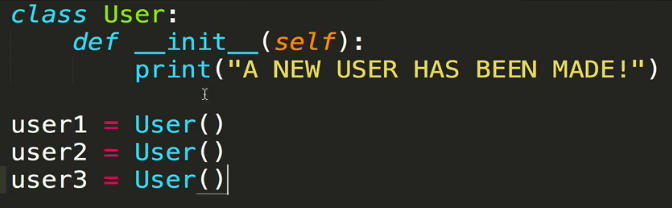


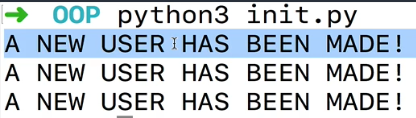




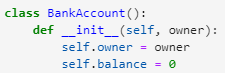


* Using the **\_\_init\_\_** method
  + Anytime you make a new instance of a class, Python will automatically look for the \_\_init\_\_ method
  + This method is called automatically, you never call it explicity. However, you have to define it
    - There are some special cases of class that hold only methods and no data (attributes), where the \_\_init\_\_ method is not needed
  + The \_\_init\_\_ method is defined and called with the *self* keyword. Self just refers to that particular instance of the class
    - Self must always be defined in the \_\_init\_\_ definition. However, we never pass it in as an argument when creating the instance. That all happens behind the scenes
    - Note that in the example below, a print statement is used to illustrate the purpose. In reality you would almost never place a print statement in \_\_init\_\_

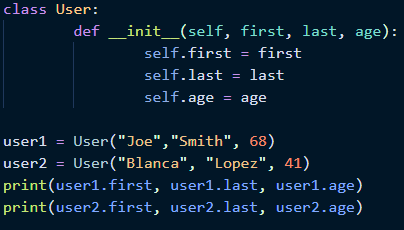




* + - The **self** keyword, again, refers to that specific instance of the user class
  + When defining \_\_init\_\_, you include the parameters that you plan on passing in to the instance in the definition
    - The \_\_init\_\_ function, when called by Python, can then access the values of those parameters to set attributes. By convention, the attribute names are usually defined as *self.attribute\_name*
      * Note that you can initialize attributes that are not based on any arguments or parameters passed into the class call

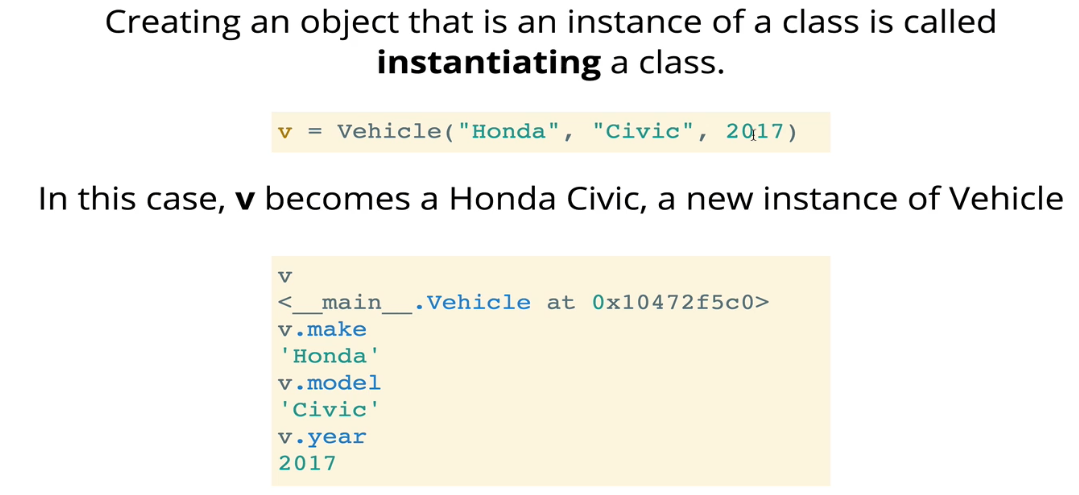


* + - You can then access those attributes by using *instance\_name.attribute\_name*

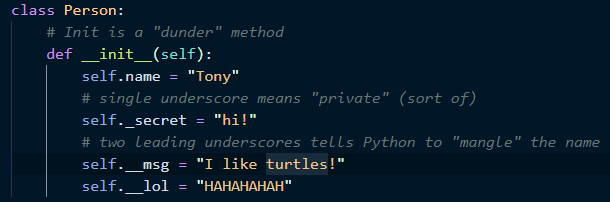




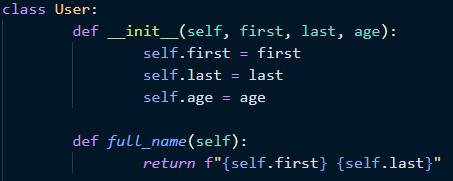
* + Another example: vehicles



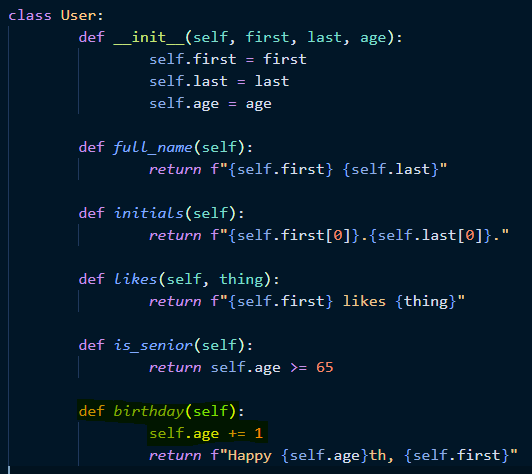
* Side topic: **underscores** and the difference between them.
  + Single preceding underscore: \_name
    - Usually used for “private” variables that are not intended to be used outside of the class. It will typically not be directly called
    - These variables are used internally only
  + Double preceding underscores: \_\_name
    - This is actually technical! It’s the only underscore variable where Python will actually do something with it; it’s not simply a convention
    - This is called **name mangling**. Behind the scenes, Python will change the name of the attribute to “\_ClassName\_\_attribute”.
      * You can still access those variables by calling them by the line above
    - It’s sole purpose is to make that attribute or method particular to the class in which it is defined
      * This will be important later when we talk about **class inheritance**, where a new class will inherit methods or attributes that may have their own unique values. The mangling ensures that the new attributes/methods do not conflict with those defined by the parent class
      * We’ll see more on this later when we talk about class inheritance
  + Dunders: \_\_name\_\_
    - Dunders are special “magic methods” that Python will look for
    - For example, the \_\_init\_\_ method is looked for when instantiating a class
    - You should refrain from defining your own dunder methods, which are generally reserved for Python-specific methods



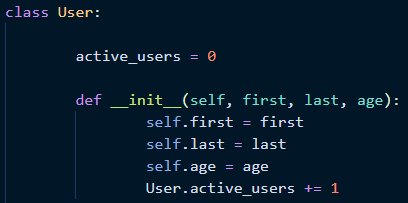
* Adding **instance** **methods**
  + There are also class methods, which are a bit different and we’ll talk about later
  + This is where the rubber hits the road and OOP becomes extremely useful
  + It is convention to define your instance methods *after* \_\_init\_\_
  + Instance methods are no different than any other function you define, except that they are defined within the class
  + However, **instance methods must always be defined with self**as the first parameter, as that is how the particular information for that instance are passed to the method. **You must define methods with self even if you are not using self within the method** (although most of the time you do us it)
    - In order to access the attributes for this instance, you have to call them with *self.attribute\_name*



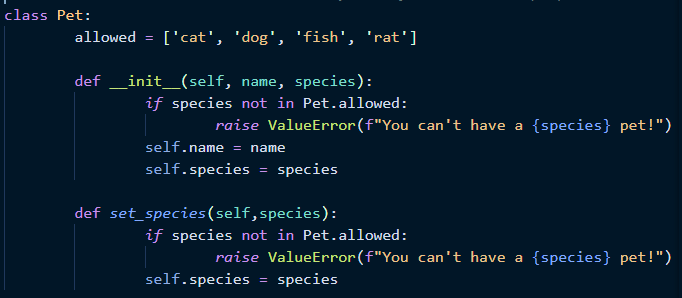
* + A **setter** method is one that edits the attributes of the instance. In the example below, the birthday() method is a setter because it modifies the age attribute



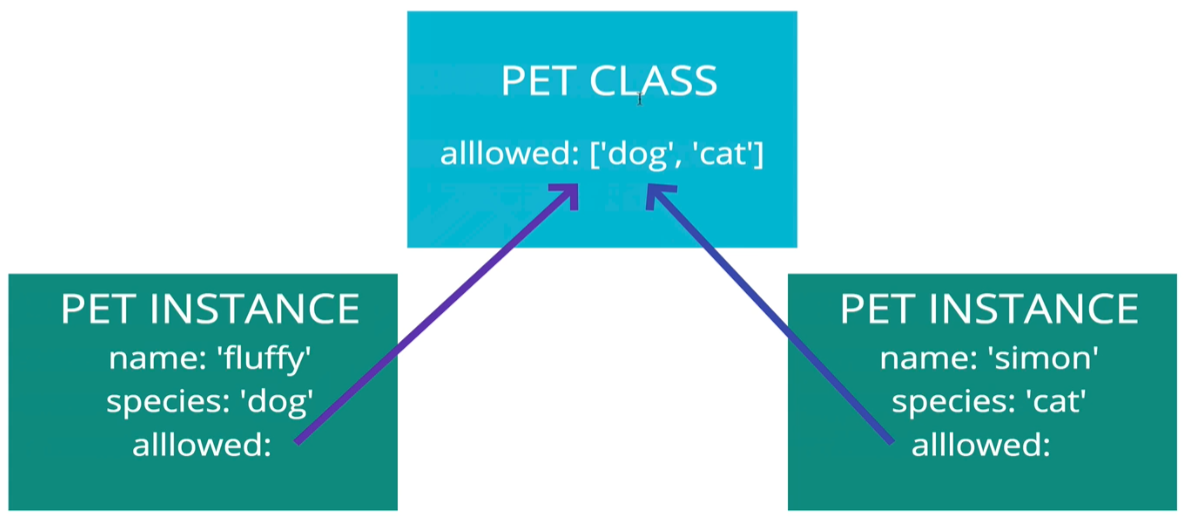
* **Class Attributes** are defined once and live on the class itself, so they are shared by all instances of that class
  + They are used far less often than instance attributes
  + To create a class attribute, simply declare it outside of any methods (conventionally at the top of the class)
  + In the example below, the class attribute active\_users is used to keep track of how many active users are in a chat room. Under the \_\_init\_\_ method, we modify the User.active\_users class attribute by adding 1 to it!
    - In the console, you can access class attributes by referring to the class name, as well as through the individual instance names (more on that later)



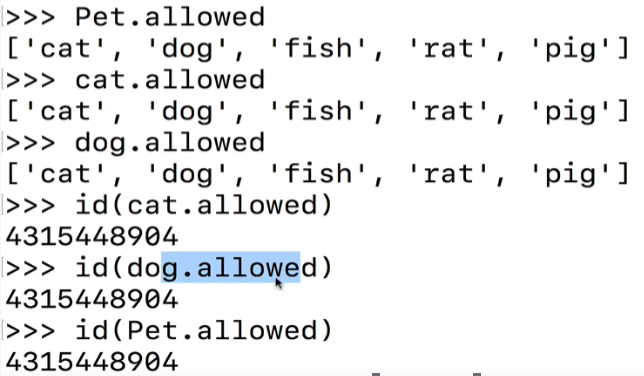
* Another common use case for class attributes is **validations**
  + That is, you can restrict the types of instances that are created for a class
  + In this example, the *\_\_init\_\_* method will check whether the species that was passed into the function when the instance was created is within the *allowed* list. If it is not, an error will be raised.
  + Additionally, a method *set\_species* exists for when one wants to change the species, where again the method checks whether the species is in the allowed list



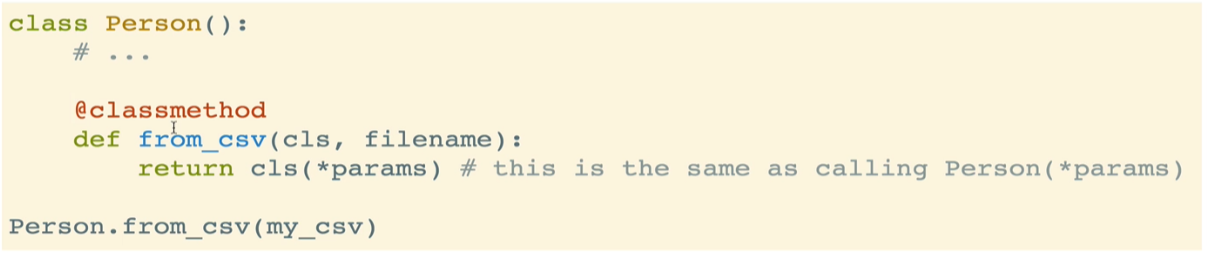
* + You can also edit the class attributes by accessing that attribute via *Class\_Name.class\_attribute*. Once you update a class attribute, all instances of that class will recognize the update. All of the instances will refer to the updated attribute when necessary (memory-based proof below)



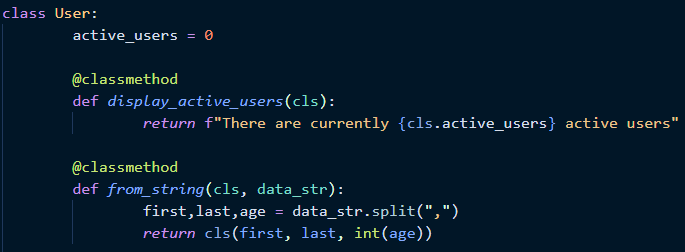
* + - When reading attributes (not writing!), because all instances refer to the same class attribute, it is often less confusing to simply read to the attribute through the class name (e.g. Class\_Name.class\_attribute). However, you can also “read” to the attributes through the instance names if you want, as they all refer to the class attribute. But that tends to be confusing



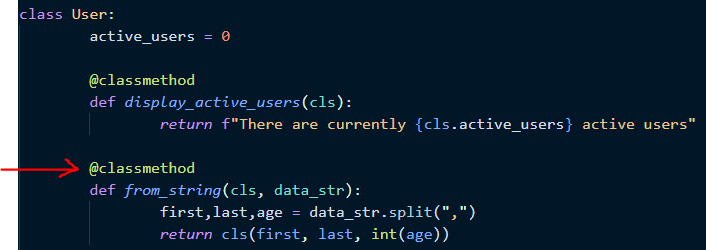
* + - Writing attributes is a different story. As soon as you edit a class attribute using an instance name (instead of a class name), that class attribute will become a unique instance attribute
* **Class methods** also exist, and are quite rare. Class methods are not concerned with instances, but the class itself
  + Class methods use the @classmethod decorator. By including this decorator before your method definition, you declare it as a class method. If you do not include the decorator, a method defaults to an instance method



* + Class methods can be defined anywhere by convention
  + The class will automatically be passed into a class method when it is called. So instead of *self*, we standardize by using *cls* (this name is not required, but is conventional)
    - cls indicates that we’ll be using the method on the class, not on the user



* + Class methods are more commonly used when creating a new instance of the class
    - For example, if you are reading in data and want to initialize a class based on that data, but it’s not in the correct format, you can create a method to format that data
    - In the example below, the method from\_string will take a single string, split it up on the commas, and then return a new instance of the class based on the split string



* The **\_\_repr\_\_** method is a way to provide a nicer string representation
  + It stands for “representation”
  + It allows us to define a representation of an instance that, when we print it, we can control what it looks like
    - Useful because it allows you to come up with nice custom representations
    - An alternative to that ugly memory object crap that defaults when you print an instance

