Unit 3-2: Objects and Classes

* What’s An Object
  + Short answer: Everything.
  + Slightly longer answer: Objects are essentially things that can be assigned to a variable and passed to a function as an argument. Everything you've used so far is an object, whether list, string, integer, dictionary, or even function.
  + In this lesson, we're going to learn how to define the "blueprint" of an object, known as a class, and how to create an object based on that blueprint through **instantiation**.
* Classes
  + As we mentioned before, you can think of classes like blueprints. Whereas a function defines a process that takes inputs and outputs, a class defines the structure of an object.
  + The set of attributes and methods defined in a class will be part of any object that is instantiated from this blueprint. We'll cover instantiation in detail as we progress through the lesson.
* Defining Classes
  + Class definitions are similar to function definitions. But, instead of starting with def as we did with functions, we use the class keyword:
    - class Person(object):
      * # code here
  + A couple of considerations to note:
    - As with functions, the definition of a class has the name followed by parentheses and a colon. The meaning of the items in the parentheses are different and refer to what the class inherits from. We'll discuss this later on.
    - Classes are conventionally named in title case, as opposed to functions, which are lowercase.
* Class Attributes
  + Classes can have class attributes or class variables defined within them. These are variables shared by all instances of the class. To define a class attribute, they must be declared outside of any methods.
    - class Person(object):
    - age = 20
    - gender = 'male'
    - friends = []
  + Remember, we're writing the blueprint of the object, but this is not the object itself.
* Using Objects
  + Now that we have our Person() class with some defined class variables, we can instantiate the class as an object.
    - class Person(object):
    - age = 20
    - gender = 'male'
    - friends = []
    - linus = Person()
  + We assigned a new variable, linus, to be an instantiation of the class Person() by using the class name like a function with the parentheses following the name. linus is now an object of type Person().
* Accessing Attributes
  + We can access linus' age and gender using dot notation, as demonstrated below.
  + linus.age
  + > 20
  + linus.gender
  + > 'male'
  + These are the class variables we defined in the class definition. Our linus object is not the class itself but rather an instance of the class. This means that this object is just one of many instances of Person() that we could create.
* Shared Data
  + Let's create another instance of our Person() class and add some friends for linus.
    - mampush = Person()
    - mampush.friends.append('Petunia')
    - mampush.friends.append('Elmer')
    - mampush.friends
    - > ['Petunia', 'Elmer']
  + What if we go back now and look at the friends of linus?
    - linus.friends
    - > ['Petunia', 'Elmer']
  + You may have thought that the friends list in linus would be empty, as mampush and linus are two separate objects. However, because we defined friends as a class variable and it is a mutable (changeable) data type, this list is shared between the different instances of the class.
  + Note that, for immutable data types such as integers or strings, changes to the class attributes will not be shared. If we change one of these for linus, the change will not appear for mampush.
    - linus.age = 30
    - linus.gender = 'female'
    - linus.age
    - > 30
    - linus.gender
    - > 'female'
    - mampush.age
    - > 20
    - mampush.gender
    - > 'male'
* Instance Attributes
  + Perhaps we want the friends lists to be separate for each person. If we want the attribute to be distinct for each instance of the class, we will need to define friends as an instance attribute, as opposed to a class attribute.
  + Let's dig a little deeper into using a class to instantiate an object, then find out how to define an instance attribute.
  + The init Method
    - As it turns out, when we instantiate an object by typing linus = Person(), the class is calling a special function, \_\_init\_\_.
    - We can redefine the initialization function, adding our own code that will run when we instantiate the object:
      * class Person(object):
      * age = 20
      * gender = 'male'
      * friends = []
      * def \_\_init\_\_(self):
      * print ('Initializing...')
    - Now, when we instantiate the class as an object, we will see this printout:
      * linus = Person()
      * > Initializing...
    - The \_\_init\_\_ method in class definitions is a special function that acts in a predetermined way. Specifically, \_\_init\_\_ is the function that runs when an instance of the class is created by calling Person(). We can explicitly add to its definition if we want to perform additional tasks when creating an instance.
* Using the Self Keyword
  + Let's take a closer look at that \_\_init\_\_ method.
  + class Person(object):
  + age = 20
  + gender = 'male'
  + friends = []
  + def \_\_init\_\_(self):
  + print ('Initializing...')
  + The first argument passed to the function, self, is required when defining methods for classes. The self argument is a reference to a future instantiation of the class. In other words, self refers to the object linus when we call linus = Person().
  + Let's add another method to our class definition: print\_stats(). This will describe the person's age and gender.
    - class Person(object):
    - age = 20
    - gender = 'male'
    - friends = []
    - def \_\_init\_\_(self):
    - print('Initializing...')
    - def print\_stats(self):
    - print('Age:', self.age)
    - print('Gender:', self.gender)
  + Notice that, once again, the first argument is self. In methods defined within a class, the self argument always comes first. Let's instantiate a person with this new code and use the function.
    - dan = Person()
    - > Initializing...
    - dan.print\_stats()
    - > Age: 20
    - > Gender: male
  + When you use an instance method like print\_stats(), you are not required to actually provide the self argument to the function — Python automatically fills in the self argument.
* Instance Variables
  + Let's return to the issue of the friends list being shared between our separate objects.
  + We want each instance of our Person() class to have non-shared friends. In other words, we want each instance to have their own friends list.
  + Instead of defining our attributes as class variables, we can define them as instance variables during initialization.
    - class Person(object):
    - def \_\_init\_\_(self, age=20, gender='male', friends=None):
    - self.age = age
    - self.gender = gender
    - if friends is None:
    - friends = []
    - self.friends = friends
    - def print\_stats(self):
    - print('Age:', self.age)
    - print('Gender:', self.gender)
  + Because we define the list through self.friends = friends, we're explicitly saying to define the list as an instance variable. Whatever friends is will be specific to the instance and therefore not shared.
  + What happens now when we change the friends variable for our objects?
    - linus = Person()
    - mampush = Person()
    - linus.friends.append('Petunia')
    - linus.friends
    - > ['Petunia']
    - mampush.friends
    - > []
  + Appending items to the friends attribute of linus does not affect the friends attribute of mampush.
* Class Variables vs. Instance Variables
  + In general, you should remember that variables assigned with self will be *instance variables*, and variables without self will be *class variables*. Which is more appropriate is up to you, but instance variables are more commonly used than class variables.
    - class Traveler(object):
    - continents = ['North America','South America','Asia','Europe',
    - 'Africa','Antarctica','Australia'] # a class variable
    - def \_\_init\_\_(self, name='Fred', visited=['Asia','Europe']):
    - self.name = name # instance variable
    - self.visited = visited # instance variable
  + In the code above, we have a class for a Traveler(). The list of continents is shared across instances, as it is a class variable, but the name and continents visited are instance variables specific to each Traveler() object created.
* Assigning New Attributes
  + Let's return to our Person() example. It's also possible to assign new attributes after an object has been instantiated. Below we add a name variable to an instance of our Person() class.
    - linus.name = "Linus"
    - linus.name
    - > "Linus"
  + The instance "Linus" has the attribute name, but note that this changes only this particular instance of our Person() class. New instances of the Person() class will not have the name attribute.
* An Example Class
  + Let's create another class that defines a building. In it we have some attributes for the average square feet, bedrooms, and bathrooms of a building. We also have two functions: one to print the attributes out in describe\_building(), and one to calculate the average price and return it based on those attributes in get\_avg\_price().
    - class Building(object):
    - avg\_sqft = 12500
    - avg\_bedrooms = 3
    - avg\_bathrooms = 2
    - def describe\_building(self):
    - print('Avg. Beds:', self.avg\_bedrooms)
    - print('Avg. Baths:', self.avg\_bathrooms)
    - print ('Avg. Sq. Ft.:', self.avg\_sqft)
    - def get\_avg\_price(self):
    - price = self.avg\_sqft\*5 + self.avg\_bedrooms\*15000 + self.avg\_bathrooms\*15000
    - return price
  + We can instantiate a Building() object using this class and use the instance methods as defined in the class.
    - average\_bldg = Building()
    - average\_bldg.avg\_sqft
    - > 12500
    - average\_bldg.get\_avg\_price()
    - > 137500
* Inheritance
  + At this point, let's return to the class definition class Building(object): to learn about another essential feature of classes: **inheritance**.
  + When we define our Building() class and specify Building(object) in the definition, we're indicating that the Building() class we're creating inherits characteristics from the object class. The built-in object class in Python is the most basic "parent class" we could have.
  + Let's say we create another class for mansions, but, instead of inheriting from object, we inherit from the Building() class we've just created:
    - class Mansion(Building):
    - avg\_sqft = 125000
    - avg\_bedrooms = 8
    - avg\_bathrooms = 10
  + In the class definition, the average square feet, bedrooms, and bathrooms have been changed, but nothing else has been done.
  + Because the Mansion() class inherits from the Building() parent class, it has access to the instance methods we defined for Building().
    - For example, we can create an instance and still use the describe\_building() function:
      * class Mansion(Building):
      * avg\_sqft = 125000
      * avg\_bedrooms = 8
      * avg\_bathrooms = 10
      * avg\_mansion = Mansion()
      * avg\_mansion.describe\_building()
      * > Avg. Beds: 8
      * > Avg. Baths: 10
      * > Avg. Sq. Ft.: 12500
    - Inheritance is an extremely powerful feature of classes. It allows us to create "generic" parent classes, such as the Building() class, and then create child classes like Mansion() that represent subsets of the parent class. We change the characteristics of the building when we define the Mansion() class, but, because it inherits from Building(), we're still able to use the parent methods describe\_building() and get\_avg\_price(). We don't need to rewrite these methods in the child class and thus make our code more concise.
  + Using inheritance, you can easily create hierarchies of functionality. This keeps your code clean and intuitive.
    - It is important to note that child classes do not affect the functionality of the parent class. Below we've added a function how\_big() to the Mansion() class.
      * class Mansion(Building):
      * avg\_sqft = 125000
      * avg\_bedrooms = 8
      * avg\_bathrooms = 10
      * def how\_big(self, bigness='very big'):
      * print('The mansion is', bigness)
    - If we create an instance of the Mansion() class, we can use the instance method and see the print statement:
      * mans = Mansion()
      * mans.how\_big(bigness='EXTREMELY big')
      * > The mansion is EXTREMELY big
    - However, because inheritance is a one-way street, if we create an instance of the Building() class, we are not able to use the how\_big() function:
      * bldg = Building()
      * bldg.how\_big()
      * AttributeError: 'Building' object has no attribute 'how\_big'
* Overwriting Inherited Attributes
  + You can also overwrite attributes and methods inherited from the parent class while defining a child class. We saw this when we overwrote the avg\_sqft, avg\_bedrooms, and avg\_bathrooms attributes in the Mansion() class.
  + Say the formula for the price of a Mansion() was different, in general, than the formula for the price of a Building(). By redefining the get\_avg\_price() method for the Mansion() class, we can change the behavior from its parent class.
  + class Mansion(Building):
  + def get\_avg\_price(self):
  + return 1000000
  + mans = Mansion()
  + bldg = Building()
  + bldg.get\_avg\_price()
  + > 137500
  + mans.get\_avg\_price()
  + > 1000000
* Def \_