1. OBJECT ORIENTED PROGRAMMING
   a. Basics:
      i. We’ve focused on designing the recipe for a program. This is the set of instructions necessary to
         accomplish a certain task. It’s the logic of a task.
      ii. This is often called: Procedural programming. The focus is on the logic of the procedures.
      iii. When programs get more complex, the logic needed to write them can get more and more twisted. It’s
           often hard to make a case for how to divide a certain task in smaller functions.
      iv. When the logic is simple, it’s easy to divide the program logically.
      v. As the programs get more complicated, what is best is to divide the problem according to the logic of the
         real world problem that the program is trying to model or represent.
      vi. Object oriented thinking helps with this - by connecting the program to the real-world.
      vii. So in object-oriented programming there is still logic and recipes, but everything is encapsulated by the
           idea of an object.
   b. What is an object?
      i. An object can be anything. It can be a person, a company, or even something very abstract like a
         geometrical shape.
   c. Object-oriented programming is a big area to explore, with lots of details, nuances, philosophies. We are not
      going to be concerned with the complicated stuff. As with the rest of the programming, we are going to be
      focused on the basics.
   d. Fundamentals of an Object
      i. An object has two types of characteristics.
         1. It has properties. For example, if you want to define a person object, then you might want to define
            a name as a property, and an age, and possibly where they live, or what they do for a living. If you
            want to define a pet object, you might want to define how many legs it has. This allows an object to
            be like a data structure - like the points we were talking about previously.
         2. It also has methods that are associated with that object. Methods are functions that are tailored
            specifically to be applied to those objects. For example, if you define the person object, that has the
            property of a name, age, and the place where they live. Then one thing that a person could do, a
            method, is that it could move towns, or get older, or change names. Methods change the properties
            of the object. They make the object do things in the program.

2. DEFINING A CLASS
   a. To define a class, we use the word class, followed by the name of your new class, and then a colon.
      class pet:
   b. Everything that is indented following the class will be contained within the class.
   c. To define the properties of a class, we simply use variables inside the class.
      class pet:
        number_of_legs = 0
   d. This is the very basics of defining a class.

3. INSTANCES OF A CLASS
   a. If we want to create an instance of that class, we have to call the class pet and assign it to a variable.
      cashew = pet()
   b. This creates an instance of that class.
   c. We can now manipulate its properties.
      cashew.number_of_legs = 4
   d. And we can use the properties of that object.
      print “Cashew has %s legs.” % cashew.number_of_legs

4. NOTICE THAT WE HAVE BEEN USING CLASSES ALL ALONG
   a. A list is a class.
   b. So is a dictionary.
   c. Whenever we import a module into Python, like math, random, turtle - we are importing a class.
5. POINT EXAMPLE
   a. In the last class we were trying to define a structure called a point. The point had two attributes: an x coordinate and a y coordinate.
   b. We can now define a class point that has these two attributes.
      ```
      class point:
          x = 0
          y = 0
      p1 = point()
      p1.x = 1.0
      p1.y = 2.0
      ```
   c. A class can have any number of properties.
   d. And the properties can be of any type.

6. METHODS
   a. So far, the objects that we can do only serve as data structures - as containers for variables. That’s all well and good, but we want to be able to create some functions that can manipulate that data in interesting ways.
   b. Methods are functions inside a class. You define a method exactly as you define a function. The only difference is that this function belongs to a class. And it only operates on objects of that class.
   c. The difference is largely conceptual.
      i. You don’t just call a function now that adds to values together. Or that calculates the factorial of a number. The idea is that functions should be associated with data structures.
      ii. If you have a list, then maybe you want to be able to sort the list. If you have a person, then maybe you want to have it move to a new place. If you have a point, then maybe you want to figure out its distance to another point. If you have a pet object, then maybe you want it to bark.
   d. There is one more difference between a method and a function.
   e. When we define a method, we need to pass as a parameter the object that we are going to operate on.
   f. If we go back to our pet example.
      ```
      class pet:
          number_of_legs = 0
          def bark(self):
              print “woof”
      pinenut = pet()
      pinenut.bark()
      ```
   g. There are several reasons why the self parameter is important. We will learn this as we go along. For now, always remember to add it.
   h. We can add as many methods as we like to a class.
      ```
      class pet:
          number_of_legs = 0
          def bark(self):
              print “woof”
          def count_legs(self):
              print “I have %s legs.” % self.number_of_legs
      pistachio = pet()
      pistachio.number_of_legs = 4
      pistachio.count_legs()
      macadamia = pet()
      macadamia.number_of_legs = 0
      macadamia.count_legs()
      ```
   i. One of the uses of the self parameter is that when Python goes to call the method, it knows which object you are trying to refer to.
   j. The main thing to remember about methods is that they are just like functions - except that they always receive a self parameter.
7. TWO SPECIAL METHODS: INIT and STR.
   a. In the last class, we not only were trying to define a structure called a point, which had two attributes: an x coordinate and a y coordinate. But we were also trying to define some operations on that point.
   b. As it is, it was all held together loosely. Now we can pull all of these things together into a tight little capsule call a class.
   c. When we call to create an instance of the class point, we can ask the user to give us certain initial properties. The method __init__ is the initializer of the class.

```python
class point:
    def __init__(self, a, b):
        self.x = a
        self.y = b

p1 = point(1.0, 2.0)
p1.x
p1.y
```

d. We can make it so that there are some default values - that way the user can still create the object even if it doesn’t know what the properties will be - like the number of legs, or the x and y.

```python
class point:
    def __init__(self, a=0, b=0):
        self.x = a
        self.y = b

p1 = point(1.0, 2.0)
p1.x
p1.y
p2 = point()
p2.x
p2.y
```
e. We can now create a method that is specific to that point class. For example, say that we want to be able to move the point by a certain x and y amount.

```python
class point:
    def __init__(self, a=0, b=0):
        self.x = a
        self.y = b
    def move(self, a, b):
        self.x += a
        self.y += b

p1 = point()
p1.x
p1.y
p1.move(1,2)
p1.x
p1.y
```
f. The second special method is the __str__ method. If we define this method, then Python knows how to convert the object into a string - which is then useful for printing purposes.

```python
class point:
    def __init__(self, a=0, b=0):
        self.x = a
        self.y = b
    def move(self, a, b):
        self.x += a
        self.y += b
    def __str__(self):
        return "(%d,%d)" % (self.x, self.y)

p1 = point()
print p1
p1.move(2,3)
```