WHAT IS OOP?

In all the programs we wrote till now, we have designed our program around functions i.e. blocks of statements which manipulate data. This is called the *procedure-oriented* way of programming. There is another way of organizing your program which is to combine data and functionality and wrap it inside something called an object. This is called the *object oriented* programming paradigm. Most of the time you can use procedural programming, but when writing large programs or have a problem that is better suited to this method, you can use object oriented programming techniques.

CLASS CONCEPT

What if you wanted to represent something much more complicated?

For example, let's say you wanted to track a number of different animals. If you used a list, the first element could be the animal's name while the second element could represent its age.

How would you know which element is supposed to be which? What if you had 100 different animals? Are you certain each animal has both a name and an age, and so forth? What if you wanted to add other properties to these animals? This lacks organization, and it's the exact need for *classes*.

Classes are used to create new user-defined data structures that contain arbitrary information about something. In the case of an animal, we could create an `Animal()` class to track properties about the Animal like the name and age.

It's important to note that a class just provides structure—it's a blueprint for how something should be defined, but it doesn't actually provide any real content itself. The `Animal()` class may specify that the name and age are necessary for defining an animal, but it will not actually state what a specific animal's name or age is.

It may help to think of a class as an *idea* for how something should be defined.

A **class** creates a new *type* where **objects** are **instances** of the class. An analogy is that you can have variables of type `int` which translates to saying that variables that store integers are variables which are instances (objects) of the `int` class.

Python Objects (Instances)

While the class is the blueprint, an *instance* is a copy of the class with *actual* values, literally an object belonging to a specific class. It's not an idea anymore; it's an actual animal, like a dog named Roger who's eight years old.

Put another way, a class is like a form or questionnaire. It defines the needed information. After you fill out the form, your specific copy is an instance of the class; it contains actual information relevant to you.

You can fill out multiple copies to create many different instances, but without the form as a guide, you would be lost, not knowing what information is required. Thus, before you can create individual instances of an object, we must first specify what is needed by defining a class.

Variables that belong to an object or class are referred to as **fields**. Objects can also have functionality by using functions that *belong* to a class. Such functions are called **methods** of the class. This terminology is important because it helps us to differentiate between functions and variables which are independent and those which belong to a class or object. Collectively, the fields and methods can be referred to as the **attributes** of that class.

Fields are of two types - they can belong to each instance/object of the class or they can belong to the class itself. They are called **instance variables** and **class variables** respectively.

THE SELF

Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list, but you **do not** give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object *itself*, and by convention, it is given the name `self`.
You must be wondering how Python gives the value for `self` and why you don't need to give a value for it. An example will make this clear. Say you have a class called `MyClass` and an instance of this class called `myobject`. When you call a method of this object as `myobject.method(arg1, arg2)`, this is automatically

converted by Python into `MyClass.method(myobject, arg1, arg2)` - this is all the special `self` is about.
This also means that if you have a method which takes no arguments, then you still have to have one argument - the `self`.

The __init__ method

There are many method names which have special significance in Python classes. We will see the significance of the `__init__` method now.
The `__init__` method is run as soon as an object of a class is instantiated (i.e. created). The method is useful to do any *initialization* (i.e. passing initial values to your object) you want to do with your object. Notice the double underscores both at the beginning and at the end of the name.

Code

```
class Player:
   """ Represent a player, with a name"""
   #A class variable, counting the number of players
   total_players = 0
   def __init__(self, name):
      self.name = name
      print('Here comes on the crease {}'.format(self.name))
      Player.total_players += 1
   def out(self):
      print('{} lost his wicket'.format(self.name))
      Player.total_players -= 1
      if(Player.total_players == 0):
         print('{} was the last man'.format(self.name))
      else:
         print('There are still {:d} players left'.format(Player.total_players))
   def say_hi(self):
      print('Greetings, my name is {}'.format(self.name))
```

```
    #class method
    @classmethod
    def how_many(cls):
        print("We have {:d} players".format(cls.total_players))

#player1 is the objecg of class Player
player1 = Player("Rohit Sharma")
player1.say_hi()
Player.how_many()
player2 = Player("Virat Kohli")
player2.say_hi()
Player.how_many()
print("There is partnership between the two players")
player1.out()
player2.out()
Player.how_many()
```