**Object Oriented Programming**

This chapter deals with a very useful technique called Object Oriented Programming (OOP). To understand OOP, consider a game that needs to track everything about the players in the game. We could set up a bunch of variables:

Player1name = “Mario”

Player2name= “Yoshi”

Player1speed = 3

Player2speed = 7

Player1Health = 9

Player2Health = 5

And so on.

The code later on would look something like this:

If player1Health == 0:

Print (“Player 1 you dead!”)

If player2Health == 0:

Print (“Player 2 you dead!”)

And so on.

The problem with this becomes evident when you have many players to track – say 100 players in a multi-player game. You would have too many variables and if statements which makes your code very long and inefficient. There is a better way.

In OOP, we create a **class** of objects. A class is a description of a type of thing. For example, we could make a Player class. Inside the class description, we include things such as name, health, speed and so on. This class would apply the same way to all players, whether you have 1 player or 1000 players. Here’s how it works:

**Classes and Objects**

A **class** is a description of a type of object. It could be anything: a player, a car, a ball. In Python we can create a class easily. Using our previous example, lets create a Player class:

class Player:

Notice the first letter of the class name is capitalized – this is a common convention. We’ll see why it makes sense to do so. Next, we initialize some properties of our class:

class Player:

health = 10

name = "Ralph"

speed = 7

The three properties – health, name and speed – are called **class** **attributes**. Each attribute is represented by a variable, sometimes called a **member** **variable**.

Now we have defined what a Player is but we haven’t created one. Here’s how we create a Player. After the above code, add these two lines:

playerOne = Player # this creates a new player

print (playerOne.name) # should print “Ralph”

Once a class is defined, we can create as many objects as we want:

playerOne = Player # this creates a new player

print (playerOne.name) # should print “Ralph”

playerTwo = Player # this creates a new player

print (playerOne.name) # should print “Ralph”

**Initializing Objects**

This is great, but it isn’t very useful. We have created two identical players. We would like to create players with different properties. To do this, we use a slightly different approach. We ***initialize*** each player using a function called \_\_init\_\_. Change your code to this:

class Player:

def \_\_init\_\_(self, h, n, s): # function to set attributes

self.health = h

self.name = n

self.speed = s # Player definition ends here

# this is the main code

player1 = Player(4, "Charlie", 6) # create a new player

player2 = Player(3, "Joe", 9) # create another new player

Here’s a brief explanation of the above code:

* The player class has 3 attributes: health, name and speed.
* The \_\_init\_\_ function is used to initialize the player’s attributes (i.e. give them starting values).
* The \_\_init\_\_ function is a special function that Python recognizes. The first parameter of the function must be self or an error will be flagged.
* Self means “this particular player”. For example, self.speed means we are talking about the player’s speed and not some other speed – for example the speed of the game or the speed of a spaceship. As we will see, all functions inside a class definition must have self as the first parameter.
* The \_\_init\_\_ function has 4 parameters in this case (self, h,n,s) but only 3 arguments are sent to it (health, name and speed).
* The underscores in the name of the function (\_\_init\_\_) are needed to make sure the function is unique. Python wants to make sure there is no other function in your program with the same name.

Now we can go ahead and test our Players by getting them to print out their names:

class Player:

def \_\_init\_\_(self, h, n, s): # function to set attributes

self.health = h

self.name = n

self.speed = s

player1 = Player(4, "Charlie", 6) # create a new player

player2 = Player(3, "Joe", 9) # create another new player

print (player1.name)

print (player2.name)

**Class Methods**

At this point we have two Players with three attributes. Now we want to give these Players some things to do. Let’s add a laugh function:

class Player:

def \_\_init\_\_(self, h, n, s): # function to set attributes

self.health = h

self.name = n

self.speed = s

def laugh(self): # function to make the player laugh

print ("Hahaha!")

# class definition ends

player1 = Player(4, "Charlie", 6) # create a new player

player2 = Player(3, "Joe", 9) # create another new player

player1.laugh()

player2.laugh()

We have created a very useless function called laugh(). Since any function inside the class describes what our player can do, a class function is called a **method**.

Notice again that the first parameter in our laugh() method is **self**. This is so the Python interpreter knows who is laughing. Again, we don’t need to send an argument to self; the Python interpreter does this automatically.

The concept of “Self” is pretty simple – we use the same concept when we say “me” or “I”. Who is “me”? It depends who is laughing:

player1.laugh() # player1 is “self”

player2.laugh() # player2 is “self”

Try changing the laugh method to this:

def laugh(self):

print (self.name+ ": "+ "Hahaha!")

Now the laugh() function prints out the name of the player that is laughing. Hopefully this makes clear the meaning of the **self** keyword.

To summarize:

* A class is a blueprint for a type of object (ex. Player is a class)
* An object is an instance of a class (ex. player1 is a Player object)
* An attribute is something the player has (ex. name). Attributes are also called member variables.
* A method is something the player does (ex. laugh()). Methods are also called member functions.
* There is one special method called \_\_init\_\_. This method is used to initialize (set up) the attributes of an object.
* All class functions (methods) must have at least one parameter called self. Self allows us to identify which object is being accessed.

Exercise:

1. Create a new method called yelp(). Yelp() will make the player say “Ouch!”. Get your players to yelp. Make sure they identify themselves:

Charlie: Ouch!

Joe: Ouch!

1. Create a method called punch(). Punch() will make the player say “Taste my fists of fury!”. Get both players to punch().
2. Create a third player, called player3. Give player3 a name, health and speed.