**Unit 5 – Object orientation**

This unit concerns itself with the concepts of object orientation, a paradigm that you may come across again and again in the programming world. Object orientation (henceforth OO) involves the use of ‘classes that contain their own data structures and functions. Once these ‘classes have been defined they can be instantiated as ‘objects’ which can be modified independently of each other.

The reasons for programming using OO mostly relate to code organisation and re-use. Consider that we want to create a program to define various persons, perhaps as part of a database. If we define a standard template for a ‘person’ all we have to do then is spin off instances of it filling in the blanks for each one. And if for any reason we wish to change the values defined in ‘person’ we only need to modify the class definition and all objects that we create from it will reflect those modifications.

The following script will achieve something to this effect.

**Person.py**

To create a new class we do the following.

class person:

This is similar to how we might define a new function, but instead we specify that ‘person’ is a class definition. We simply end the statement with a colon and as always everything contained within the class definition is indented one tab level.

All classes need to have a constructor function. This is run when an object is created from the class definition and allows us to define initial values for the object. The function is given the same name every time and should come at the top of your class definition.

def \_\_init\_\_(self, f\_name, l\_name, gender, height, weight):

self.f\_name = f\_name

self.l\_name = l\_name

self.gender = gender

self.height = height

self.weight = weight

self.species = "Human"

This constructor will allow us to set the name, gender, height and weight of person objects we create from this class. Note the ‘self’ given as an argument. This is basically a reference to the current object instance and is a required argument in all function definitions in classes. You’ll notice it in the class variables too. All references to class variables will need ‘self.’ prepended; otherwise it’ll throw an error.

We can also define pre-determined variables in the constructor, such as ‘self.species’ which will always be human in this case.

Now we can create functions that output or change values. Here we will create a function to display a person’s information and another to modify the person’s weight value.

def display\_info(self):

print("\nName: {} {}\nGender: {}".format(self.f\_name, self.l\_name, self.gender))

print("Height: {}cm\nWeight: {}kg".format(self.height, self.weight))

print("Species: {}".format(self.species))

def modify\_weight(self, value):

self.weight += value

I should point out that while ‘modify\_weight()’ seems to only add to a person’s weight, we can actually take away weight by giving the function a negative number. ‘modify\_weight(-10)’ will remove 10 from a person’s weight. This way keeps the program simpler since we don’t have to make two separate functions for increases and decreases.

Now that we have a complete class definition for person (at least for now) we can create an object, or objects, from it. We will create two here. Type the following after and outside of ‘person’.

david = person("David", "Stephenson", "Male", 180, 80)

nick = person("Nick", "Wood", "Male", 177, 66)

Here we have two people, david and nick. They both have their own set of details and are distinct entities following the same template. We can invoke some of the functions we created to see that we can access each person’s details quite easily.

david.display\_info()

david.modify\_weight(-10)

david.display\_info()

For david we can output his information, modify it before outputting it again to see the changes. We could do the same for nick if we wanted, but instead we will simply output his full name by referencing his name variables.

print("\n{} {}".format(nick.f\_name, nick.l\_name))

There are a few things to point out about functions in classes. First of all, they’re often referred to as ‘methods’ to highlight that they are functions of classes. Secondly, class functions are only available in the scope of its parent class so unless you prepend the object’s name (such as ‘david’ or ‘nick’ in the above example) then python will think you are calling an undefined function. I should also mention that class variables are sometimes referred to as ‘properties’.

**Inheritance**

Classes tend to be a good way to handle parent-child entity relationships. We can start with a fairly broad entity, like a vehicle, before breaking it down into more specific entities such as cars, planes and ships. These can even be broken down further if needs be. Let’s go through an example to demonstrate this.

**Vehicles.py**

We’ll start by setting out all the classes we intend to build.

class vehicle:

class car(vehicle):

class aeroplane(vehicle):

class jet\_fighter(aeroplane):

Here we have a class ‘vehicle’ which is extended by ‘car’ and ‘aeroplane’. ‘aeroplane’ itself is extended again by ‘jet\_fighter’. These ‘class extensions’ inherit the properties of the parent classes, so ‘jet\_fighter’ will include any properties or methods defined inside ‘aeroplane’ and ‘vehicle’. We will demonstrate this once we have filled out each class. Notice also that class extensions require the parent class passed to it as an argument.

We’ll start with ‘vehicle’ which will contain very generic information that any vehicle will have: make, model and colour. Inside ‘vehicle’ add this constructor.

def \_\_init\_\_(self, make, model, colour):

self.make = make

self.model = model

self.colour = colour

We will now add some simple methods, one to print out basic information and another to allow colour changes.

def print\_info(self):

print("\nVehicle:\t{} {} in {}".format(self.make, self.model, self.colour))

def modify\_colour(self, new\_colour):

self.colour = new\_colour

Now that we have completed ‘vehicle’ we can move on to the first class extension, ‘car’. Pay close attention to how the constructor is put together.

def \_\_init\_\_(self, make, model, colour, car\_type, top\_speed, accel):

vehicle.\_\_init\_\_(self, make, model, colour)

self.car\_type = car\_type

self.top\_speed = top\_speed

self.accel = accel

The constructor of the extended class needs to call the constructor of the parent class too. We take all the arguments required by the parent and pass them to the ‘vehicle’ constructor at the top before assigning the child properties below to the class extension.

We can also call ‘vehicle’ methods in ‘car’. This is demonstrated in the next method which outputs information like before. Below the constructor type the following.

def print\_info(self):

vehicle.print\_info(self)

print("Type:\t\t{}\nTop speed:\t{}km".format(self.car\_type, self.top\_speed))

print("Accel:\t\t0-100kmh in {} seconds".format(self.accel))

At the top of the method we call ‘print\_info()’ from ‘vehicle’ which will output the basic information of the ‘car’ while the following print statements output ‘car’ specific information. It’s also worth mentioning that we are able to have a class extension with the same method names as its parent. When we call this method with ‘my\_car.print\_info()’ (‘my\_car’ being an object instance of ‘car’) python will attempt to find ‘print\_info()’ in ‘car’ first, then ‘vehicle’ if it doesn’t find it. Likewise, ‘my\_car.modify\_colour(“blue”)’ will work since the method exists in the parent class ‘vehicle’.

Since the next two class extensions are similar to ‘car’ I’ll simply place them both here, though there is something I will point out afterwards.

class aeroplane(vehicle):

def \_\_init\_\_(self, make, model, colour, plane\_type, top\_speed, max\_range):

vehicle.\_\_init\_\_(self, make, model, colour)

self.plane\_type = plane\_type

self.top\_speed = top\_speed

self.max\_range = max\_range

def print\_info(self):

vehicle.print\_info(self)

print("Type:\t\t{}\nTop speed:\t{}km".format(self.plane\_type, self.top\_speed))

print("Max range:\t{}km".format(self.max\_range))

class jet\_fighter(aeroplane):

def \_\_init\_\_(self, make, model, colour, top\_speed, max\_range, ammo, hardpoints):

aeroplane.\_\_init\_\_(self, make, model, colour, "Jet fighter", top\_speed, max\_range)

self.ammo = ammo

self.hardpoints = hardpoints

def print\_info(self):

aeroplane.print\_info(self)

print("Ammo:\t\t{}\nHardpoints:\t{}".format(self.ammo, self.hardpoints))

You might notice here that we can even extend class extensions, though I wouldn’t recommend going any further than what we have here if you can help it. In the constructor of ‘jet\_fighter’ we call its parent constructor with ‘plane\_type’ pre-set since that’s as precise as we need. Of course, there are further sub-categories of jet fighter that you could account for, but we will keep things simple here. You may make change to this later if you like.

After extending twice, ‘jet\_fighter’ needs to be given enough arguments to satisfy its parent and grandparent constructors as well as its own. Next we’ll create some objects from these classes and you’ll see that the statements can get a little long. Type the following at the bottom of the script.

red\_car = car("Nissan", "GT-R", "red", "Sports car", 311, 2.7)

white\_plane = aeroplane("Boeing", "777", "white", "Airliner", 945, 15844)

grey\_fighter = jet\_fighter("Eurofighter", "Typhoon", "grey", 2495, 2900, 150, 13)

Now we have created some objects using these class extensions. We could also make a ‘vehicle’ object but it would have sparse information.

Add the following to make sure the script works. We’ll print out information for each object and change the colour of the car to demonstrate that we can easily access parent methods.

red\_car.print\_info()

white\_plane.print\_info()

grey\_fighter.print\_info()

red\_car.modify\_colour("blue")

red\_car.print\_info()

Hopefully the output from this will give you a better idea of what is going on. You might even want to create some more class extensions for lorries, ships, helicopters, or whatever else you can think of.