Chapter 2
Objects, classes and factories

By the end of this chapter you will have the essential knowledge to start our big project – writing the MyPong application. This chapter is important for another reason – you will be introduced to a particular way of programming called **object-oriented programming (OOP).** This is one way of coping with the complexity of writing big programs. You are not going to learn everything there is to know about OOP but you will gain a foundation and understand how classes and methods can be built.

There is far more in this chapter about how to design programs than about actually writing them. After you have finished working your way through this book, you will hopefully be starting to think about how to design your own programs rather than just writing the ones that are suggested. The bonus chapter illustrates how well-designed code can quickly be adapted to make new applications.

In this chapter you are going to:

- learn how to design classes
- learn how to make objects from classes
- start to build your own module
- learn how to design larger programs one bit at a time.

Big programs

If we sat down and wanted to build ourselves a car (a real one, as in Figure 2.1) we might not even know where to start. How about starting with the engine? An engine manufacturer does not need to know anything about wheels or windscreen wipers or how to make seats – the engine manufacturer makes engines. A car manufacturer does not need to know how an engine is made; they just need to know how to attach a car to it. A car builder can also get a wheel from one company and a tyre from another company – as long as they fit.

Figure 2.1 McLaren 12C – a very pretty car!

A complex computer program, such as an email application or a video game, can also be very difficult to build because it is also made of many parts. However, just like the car, large applications do not have to be built all at once. The latest video games are often made up of millions of lines of code. If they were built as one program, can you imagine how hard it would be to find a typing mistake? And believe me, everyone makes typing mistakes. The way this problem is avoided is by building little bits at a time.

There are several ways to do this. One of these is to use OOP, which builds programs out of coded objects! The great thing about this is that if you build the **objects** in the correct

way, you can make libraries of these objects. These objects can be re-used again in other applications. You can of course do a lot of this with functions but learning about objects and **classes** will help you better understand other people's code. The popular `PyGame` library and **tkinter** both use classes.

Classes

Our first task is to look at a problem and work out how it can be separated into smaller tasks and objects. You have just learned that we build libraries of objects. This is not entirely true. We build libraries of classes. A class is better than an object, because it can act as a **factory** for making objects.

Instead of explaining with a complex situation, let's take something very simple – a cat.

There are many ways to build a Cat class. Here is one:

Code Box 2.1

```
class Cat:
    def speak(self):
        print("Meow!")

    def drink(self):
        print("The cat drinks its milk.")
        print("The cat takes a nap.")
```

Open a new window in IDLE and type in the code from Code Box 2.1. Save this as `cat.py` in your Python folder.

Analysis of Code Box 2.1

The first thing to note is that we use the `class` keyword and that the class name, `Cat`, starts with a capital. The indented code after the colon belongs to the class. Inside the class we have provided two methods. Methods are very similar to functions except that they belong to a class.

Remember, this is like a factory. Now, you will learn how to send it orders to build some objects – in this case, some cats!

Delving Deeper

What is the difference between a method and a function?

A **method** is just a special function that we make in a class.

Computer scientists might get upset with us calling classes 'factories' as this word has another meaning in advanced OOP. They would prefer us to describe classes as *templates* or *blueprints*. You will find it a lot easier to think of classes as factories though.

If you think of classes as factories, you can understand that by requiring a method to have `self` as an argument, you tell the computer that the method will be available to each of the objects made by the factory.

Modules

By storing class files such as `cat.py` in one folder they become a module! Any other Python files in the same folder can use them with a simple import **statement**:

```
import cat
```

There are other special modules in the Python library that can be used. You have already done this once.

Quick Quiz 2.1

Can you remember which module we used before?

main.py

So far we cannot do anything with the Cat class. Have you tried to run it? Not a lot happens does it?

Open a new window in IDLE and type in the code from Code Box 2.2. Then save this new file as `main.py`. Notice `cat.py` is imported at the beginning of the code which gives our program access to the Cat class. The file that you have just created is where you will start to run the program, which is why it is called `main.py`.

Code Box 2.2

```
import cat

# create an instance of a cat, named Romeo
romeo = cat.Cat()

# play with Romeo
romeo.speak()
romeo.drink()
```

Analysis of Code Box 2.2

```
romeo = cat.Cat()
```

This is where we order an object from our cat factory. Computer scientists say: "We have made an **instance** of the Cat class." They might even say that we have **instantiated** a cat object. Romeo is the object and it is created by the Cat class (hence the capital letter). The Cat class is in `cat.py` so we tell `main.py` where to look with the dot operator. The dot is used to link the class with its location. In non-coding language, this line of code translates as:

> 'Create an object called Romeo using the cat factory found in the `cat.py` file.'

Thus we have a cat! And he is called Romeo! All done with one line of code.

Romeo has all the methods that were built by its factory, the Cat class, available to him. To access them we call them using the dot operator again:

```
romeo.speak()  # calls romeo's speak method
romeo.drink()  # calls romeo's drink method
```

If you have not played with Romeo yet, you can do so now! To play with Romeo you simply save and run `main.py`. It must be in the same folder as `cat.py`.

Improving the Cat class

Warning: There are a lot of selfs in this section!

When calling a function we send it arguments in the brackets like this:

```
times_tables(12, 12)
```

This function is from *Python Basics* and calling it would print out the 12 times table up to $12 \times 12 = 144$. Objects can do this too. We could supply a name for example:

```
romeo = cat.Cat("Romeo")
```

Our Cat class will now need to be re-written though. This is not done in quite the same way as it is in a function. It is done in a special method called a **constructor**.

Constructors have a special bit of code that you will see quite often from now on. It looks a little bit frightening at first but it is always the same – you will get used to it:

```
def __init__(self, name):
```

It always has the `self` argument to make sure that everyone understands that this method is going to be available to each object built by this class. Next we must list the other arguments that we want to pass to the constructor. In the above example the other argument needed is `name`.

Now let's pause. You may not have been confused by the `init` surrounded by two underscores on each side of it; you may have noticed that this is actually a method because it has `def` at the beginning and a colon at the end; you may be a very clever coder! Most students need a little encouragement at this stage – listen to Mr Campbell.

Now back to work! In this special method called a constructor we have to create a `self.name` variable from `name` (yes that's right, so that it is available to the object created by this class). We then will use this new variable, `self.name`, in our methods.

We are adding `self.` to the front of all the variable names that are going to be available to our objects and not simply to the class. This is why we end up with a lot of `self`s!

Delving Deeper

When we build classes, we write methods that will be available to the objects we build (create instances of). In a game of snooker, there may be several balls with different locations. Each one has a `find_location()`

method that refers only to itself, even though it has been built from the same class. This is why the methods in our class have `self` passed to them as an argument. It is even cleverer than this though.

Programmatically the objects do, in actual fact, refer back to the class code for the method but each object keeps track of its own data. It is as if each object has been built with its own methods that are specific to it alone. It is as if the methods act independently of the other objects built by the same factory. This is just what we would wish for when we build objects, and it is all done with one argument – `self`.

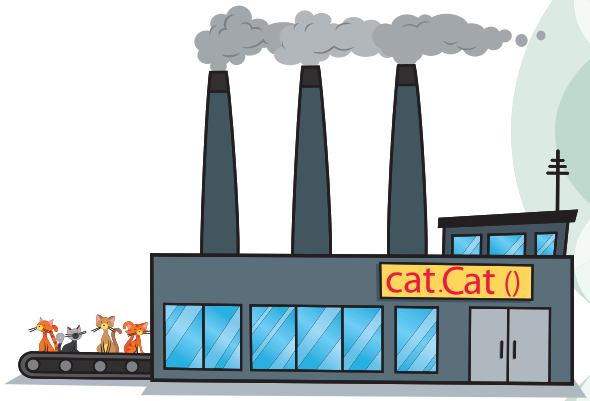
Open a new window in IDLE and copy the code from Code Box 2.3. Save it as `cat2.py`. Compare this with `cat.py` (Code Box 2.1) to see how we use the new variable `self.name` created in the constructor. Yes, that is all this constructor code has done: it has given you a new variable that you can use! As before, this class does not run anything. It is just a factory, but you are about to see how useful factories are.

Code Box 2.3

```
class Cat:
    # constructor:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(self.name, " says Meow")

    def drink(self):
        print(self.name, " drinks some milk.")
        print(self.name, " takes a nap.")
```



Improving main.py

This is where all that hard work pays off. You are going to order two cats very quickly and play with both. You will probably find the code makes sense without any help this time. Open a new window in IDLE and copy the code from Code Box 2.4. Save it as `main2.py`.

Code Box 2.4

```
import cat2

# create two instances of a cat
romeo = cat2.Cat("Romeo")
juliet = cat2.Cat("Juliet")

# play with Romeo
romeo.speak()
romeo.drink()

# play with Juliet
juliet.speak()
juliet.drink()
```

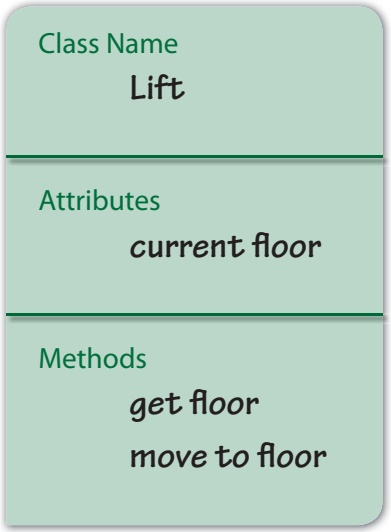
Run this file and then feel free to play and adjust the code. Have fun!

Designing classes

This section introduces another simple example where you will get to practise what you have learned and you will start to learn how to design a class. You are going to build a lift and a lift operator.

Below is one way to make a class for a lift. There are many properties a lift can have, but in terms of a program there are very few essentials.

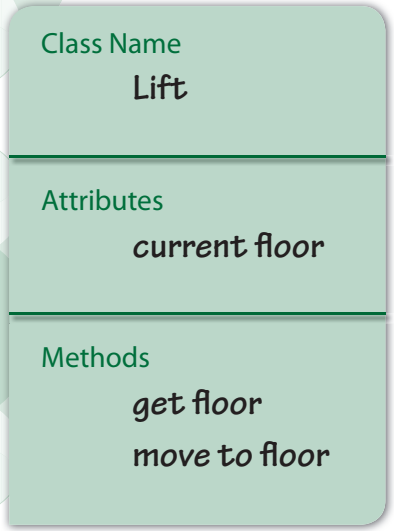
The only essential feature of a lift is which floor it is on. Instead of 'essential feature' we could call this a property or characteristic. In programming we call these characteristics **attributes** (or sometimes **parameters**) and we **initialise** them in the constructor.



Class design sheet 2.1 A lift class.

We might also want to know how many passengers it can hold, or its velocity – but this lift is going to be very simple. To plan a lift class we can use a class design sheet. You will find a blank one in the Appendix at the end of this book and also on the companion website – so you can print out your own should you wish. The design work goes into the class design sheet. From this we can build our class.

Two methods are essential: one that finds out what floor the lift is on and another to move the lift to a new floor. Now let's see how this gets turned into code:



Code Box 2.5

```
# lift.py provides a Lift Class

class Lift:
    # constructor:
    def __init__(self, current_floor=0):
        self.current_floor = current_floor

    def get_floor(self):
        return self.current_floor

    def move_to_floor(self, floor_number):
        self.current_floor = floor_number
```

Class Name

Constructor

Method

Analysis of Code Box 2.5

There is a **default value** set for the `current_floor` attribute, `0`. This can be seen in the constructor method. Now, if we do not pass a `current_floor` argument to this class when it builds a new lift object, the new lift will start on floor `0`.

The `return` key word is used in the first method. This means that if we call this method in our lift operator program it will return (give back) the value stored in the `self.current_floor` variable.

The lift operator

`lift_operator.py` has some easy code that plays with the lift. Instead of typing it up you can simply open and run it from the Chapter 2 folder in the source code files you have downloaded from the companion website. The code can be seen on the next page in Code Box 2.6.

Code Box 2.6

```
# lift_operator.py

import lift

# create a lift object
my_lift = lift.Lift()

# Find out what floor the lift is on
floor = my_lift.get_floor()
print("The lift is on floor", floor)

# move the lift to a new floor
my_lift.move_to_floor(5)

# Find out what floor the lift is on now
floor = my_lift.get_floor()
print("The lift has now moved to floor", floor)
```

Experiment

When creating `my_lift` we did not include a `current_floor` variable so the lift started on floor `0`. Try altering this line of code to `my_lift = lift.Lift(3)` and running again. Interesting, huh?

Chapter summary

In this chapter you have learned:

- about classes, objects and how to design them
- how to build your own module
- a little about object-oriented programming – OOP
- what a nuisance `self` can be!

This has been quite an intense chapter. If you are feeling overwhelmed, do not worry as the MyPong project is going to reinforce many of these ideas and you will get the hang of them soon.

If you want some practise, try this easy and relaxing challenge. (Easy and relaxing if you have the code for cat2.py and main2.py in front of you while you complete the challenge!)

Good luck.

Easy and relaxing challenge

Make a Pet class and save it in a file called `pet.py`.
Make an application called `pet_owner.py` to build and control some pets.
You can choose any pets you like such as hamsters or chinchillas. Alternatively you might prefer tarantulas or piranhas. You will find that you have to be careful with what your pets can do. Piranhas cannot speak or go for walk!