**Object-oriented programming**

Introduction

As you have seen from the earliest code examples in this course, it is not compulsory to organise your code into classes when you program in Python. You can use functions by themselves, in what is called a procedural programming approach. However, while a procedural style can suffice for writing short, simple programs, an object-oriented programming (OOP) approach becomes more valuable the more your program grows in size and complexity.

The more data and functions comprise your code, the more important it is to arrange them into logical subgroups, making sure that data and functions which are related are grouped together and that data and functions which are not related don’t interfere with each other. Modular code is easier to understand and modify, and lends itself more to reuse – and code reuse is valuable because it reduces development time.

As a worst-case scenario, imagine a program with a hundred functions and a hundred separate global variables all in the same file. This would be a very difficult program to maintain. All the variables could potentially be modified by all the functions even if they shouldn’t be, and in order to pick unique names for all the variables, some of which might have a very similar purpose but be used by different functions, we would probably have to resort to poor naming practices. It would probably be easy to confuse these variables with each other, since it would be difficult to see which functions use which variables.

We could try to make this code more modular even without object orientation. We could group related variables together into aggregate data structures. In the past, some other languages, like C++, introduced a struct type which eventually became indistinguishable from a class, but which initially didn’t have any methods – only attributes. This allowed programmers to construct compound variables out of many individual variables, and was the first step towards object orientation. In Python, we often use dictionaries for ad-hoc grouping of related data.

We could also split up the functions and data into separate namespaces instead of having them all defined inside the same global namespace. This often coincides with splitting the code physically into multiple files. In Python we do this by splitting code up into modules.

The main additional advantage of object orientation, as we saw in the previous chapter, is that it combines data with the functions which act upon that data in a single structure. This makes it easy for us to find related parts of our code, since they are physically defined in close proximity to one another, and also makes it easier for us to write our code in such a way that the data inside each object is accessed as much as possible only through that object’s methods. We will discuss this principle, which we call encapsulation, in the next section.

Some people believe that OOP is a more intuitive programming style to learn, because people find it easy to reason about objects and relationships between them. OOP is thus sometimes considered to be a superior approach because it allows new programmers to become proficient more quickly.

Basic OOP principles

The most important principle of object orientation is encapsulation: the idea that data inside the object should only be accessed through a public interface – that is, the object’s methods.

The age function we saw in the previous chapter is a good example of this philosophy. If we want to use the data stored in an object to perform an action or calculate a derived value, we define a method associated with the object which does this. Then whenever we want to perform this action, we call the method on the object. We consider it bad practice to retrieve the information from inside the object and write separate code to perform the action outside of the object.

Encapsulation is a good idea for several reasons:

* the functionality is defined in one place and not in multiple places.
* it is defined in a logical place – the place where the data is kept.
* data inside our object is not modified unexpectedly by external code in a completely different part of our program.
* when we use a method, we only need to know what result the method will produce – we don’t need to know details about the object’s internals in order to use it. We could switch to using another object which is completely different on the inside, and not have to change any code because both objects have the same interface.

We can say that the object “knows how” to do things with its own data, and it’s a bad idea for us to access its internals and do things with the data ourselves. If an object doesn’t have an interface method which does what we want to do, we should add a new method or update an existing one.

Some languages have features which allow us to enforce encapsulation strictly. In Java or C++, we can define access permissions on object attributes, and make it illegal for them to be accessed from outside the object’s methods. In Java it is also considered good practice to write setters and getters for all attributes, even if the getter simply retrieves the attribute and the setter just assigns it the value of the parameter which you pass in.

In Python, encapsulation is not enforced by the language, but there is a convention that we can use to indicate that a property is intended to be private and is not part of the object’s public interface: we begin its name with an underscore.

It is also customary to set and get simple attribute values directly, and only write setter and getter methods for values which require some kind of calculation. In the last chapter we learned how to use the property decorator to replace a simple attribute with a method without changing the object’s interface.