Chapter 21
Why Bother with Object Orientation?

21.1 Introduction

The pervious four chapters have introduced the basic concepts behind object orientation, the terminology and explored some of the motivation. This chapter looks at how object orientation addresses some of the issues that have been raised with procedural languages. To do this it looks at how a small extract of a program might be written in a language such as C, considers the problems faced by the C developer and then looks at how the same functionality might be achieved in an object-oriented language such as Python. Do not worry too much about the syntax you will be presented with; it is mostly a form of pseudo code and it should not detract from the legibility of the examples.

21.2 The Procedural Approach

Consider the following example:

```
record Date {
  int day
  int month
  int year
}
```

This defines a data structure for recording dates. There are similar structures in many procedural languages such as C, Ada and Pascal.

So, what is wrong with a structure such as this? Nothing, apart from the issue of visibility? That is, what can see this structure and what can update the contents of

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_21

the structure? Any code can directly access and modify its contents. Is this problem? It could be, for example, some code could set the day to −1, the month to 13 and the year to 9999.

As far as the structure is concerned the information it now holds is fine (that is day = 01, month = 13, year = 9999). This is because the structure only knows it is supposed to hold integers; it knows nothing about dates per se. This is not surprising, it is only data.

21.2.1 Procedures for the Data Structure

This data is associated with procedures that perform operations on it. These operations might be to

- test whether the date represents a date at a weekend or part of the working week.
- change the date (in which case the procedure may also check to see that the date is a valid one).

For example:

```
is_day_of_week(date)
in_month(date, 2)
next_day(date)
set_day(date, 9, 3, 1946)
```

How do we know that these procedures are related to the date structure we have just looked at? By the naming conventions of the procedures and by the fact that one of the parameters is a data (record).

The problem is that these procedures are not limited in what they can do to the data (for example the setDay procedure might have been implemented by a Brit who assumes that the data order is day, month and year. However, it may be used by an American who assumes that date order is month, day, year. Thus the mean of `set_day(date, 9, 3, 1946)` will be interpreted very differently. The American views this as the 3rd of September 1946, while the Brit views this as the 9th of March, 1946. In either case, there is nothing to stop the date record being updated with both versions. Obviously the `set_day()` procedure might check the new date to see it was legal, but then again it might not. The problem is that the data is naked and has no defense against what these procedures do to it. Indeed, it has no defense against what any procedures that can access it, may do to it.

21.2.2 Packages

One possibility is of course to use a package construct. In languages such as Ada, packages are common place and are used as a way of organising code and restricting visibility. For example,

```
package Dates is
    type Date is ....
    function is_day_of_week(d: Date) return Boolean;
    function in_month(d: Date, m: Integer) return
                                    Boolean;
...
```

The package construct provides some ring fencing of the data structure and a grouping of the data structure with the associated procedures. In order to use this package a developer must import the package. They can then access the procedures and work with data of the specified type (in this case Date).

There can even be data that is hidden from the user within a private part. This therefore increases the ability to encapsulate the data (hide the data) from unwelcome attention.

21.3 Does Object Orientation Do Any Better?

This is an important question "*Does object orientation do any better?*" than the procedural approach described above? We will first consider clases then inheritance.

21.3.1 Packages Versus Classes

It has been argued (to me at least) that an Ada package is just like a class. It provides a template from which you can create executable code, it provides a wall around your data with well-defined gateways etc. However, there are a number of very significant differences between packages and classes.

Firstly, packages tend to be larger (at least conceptually) units than classes. For example, the TextIO package in Ada is essentially a library of textual IO facilities, rather than a single concept such as the class `string` in Python. Thus packages are not used to encapsulate a single small concept such as `string` or `Date`, but rather a whole set of related concepts (as indeed they are used in Python). Thus, a class is a finer level of granularity than a package.

Secondly, packages still provide a relatively loose association between the data and the procedures. An Ada package may actually deal with very many data structures with a wide range of methods. The data and the methods are related primarily via the related set of concepts represented by the package. In contrast a class tends to closely relate data and methods in a single concept. Indeed, one of the guidelines relating to good class design is that if a class represents more than one concept, then you should split it into two classes.

Thus, this close association between data and code means that the resulting concept is more than just a data structure (it is closer to a concrete realisation of concept). For example:

```
class Date:

    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year

    def is_day_of_week(self):
        """Check if date is a week day"""
        # ... To be defined

    def in_month(self, month_index):
        """Check if month is in month_index"""
        return self.month == month_index
```

Anyone using an instance of `Date` now gets an object which can tell you whether it is a day of the week or not and can hold the appropriate data. Note that the `is_day_of_week()` method takes no parameters other than `self`, it doesn't need to as it and the date information are part of the same thing. This means that a user of a `Date` object will never need to get their hands on the actual data holding the date (i.e. the integers `day`, `month` and `year`). Instead, they should go via the methods. This may only seem a small step, but it is a significant one, nothing outside the object should need to access the data within the object. In contrast the data structure in the procedural version, is not only held separately to the procedures, the values for `day`, `month` or `year` it must also be modified directly.

For example, compare the differences between an excerpt from a program to manipulate dates (using a procedural programming language):

```
d: Date;
setDay(d, 28);
setMonth(d, 2);
setYear(d, 1998);
isDayOfWeek(d);
inMonth(d, 2);
```

Note that it was necessary to first create the data and then to set the fields in the data structure. Here we have been good and have used the interface procedures to do this. Once we had the data set up we could then call methods such as `IsDayOfWeek` and `InMonth` on that data.

In contrast the Python code uses a constructor to pass in the appropriate initialisation information. How this is initialised internally is hidden from the user of the class `Date`. We then call method such as `is_day_of_week()` and `is_month(12)` directly on the object date.

The thing to think about here is where would code be defined?

```
date = Date(12, 2, 1998)
date.is_day_of_week()
date.in_month(12)
```

21.3.2 Inheritance

Inheritance is a key element in an object-oriented language allowing one class to inherit data and methods from another.

One of the most important features of inheritance (ironically) is that it allows the developer to get inside the encapsulation bubble in limited and controlled ways.

This allows the sub-class to take advantage of internal data structures and methods, without compromising the encapsulation afforded to objects. For example, let us define a subclass of the class `Date`:

```
class Birthday(Date):
    name = ''
    age = 0
    def is_birthday():
        # ... Check to see if it is their birthday
```

The method `is_birthday()` could check to see if the current date, matched the birthday represented by an instance of `Birthday` and return true if it does and false if it does not.

Note however, that the interesting thing here is that not only have we not had to define integers to represent the date, nor have we had to define methods to access such dates. These have both been inherited from the parent class `Date`.

In addition, we can now treat an instance of `Birthday` as either a `Date` or as a `Birthday` depending on what we want to do!

What would you do in languages such as C, Pascal or Ada? One possibility is that you could define a new package Birthday, but that package would not extend Dates, it would have to import Dates and add interfaces to it etc? However, you certainly couldn't treat a Birthday package as a Dates package.

In languages such as Python, because of polymorphism, you can do exactly that. You can reuse existing code that only knew about Date, for example:

```
birthday = Birthday(12, 3, 1974)

def test(date):
    # Do something that works with a date

t.test(birthday)
```

This is because `birthday` is indeed a type of `Date` as well as being a type of `Birthday`.

You can also use all of the features defined for Date on Birthdays:

```
birthday.is_day_of_week()
```

Indeed, you don't actually know where the method is defined. This method could be defined in the class `Birthday` (where it would override that defined in the class `Date`). However, it could be defined in the class `Date` (if no such method is defined in `Birthday`); without looking at the source code there is no way of knowing!

Of course, you can also use the new methods defined in the class `Birthday` on instance (objects) of this class. For example:

```
birthday.is_birthday()
```

21.4 Summary

Classes in an object-oriented language provide a number of features that are not present in procedural languages. To summarise, the main points to be noted from this chapter on object orientation are:

- Classes provide for inheritance.
- Inheritance provides for reuse.
- Inheritance provides for extension of a data type.
- Inheritance allows for polymorphism.
- Inheritance is a unique feature of object orientation.