Chapter 18
Introduction to Object Orientation

18.1 Introduction

This chapter introduces the core concepts in Object Orientation. It defines the terminology used and attempts to clarify issues associated with hierarchies. It also discusses some of the perceived strengths and weaknesses of the object-oriented approach. It then offers some guidance on the approach to take in learning about objects.

18.2 Classes

A class is one of the basic building blocks of Python. It is also a core concept in a style of programming known as Object-Oriented Programming (or OOP). OOP provides an approach to structuring programs/applications so that the data held, and the operations performed on that data are bundled together into classes and accessed via objects.

As an example, in an OOP style program, employees might be represented by a class Employee where each employee has an id, a name, a department and a desk_number, etc. They might also have operations associated with them such as `take_a_holiday()` or `get_paid()`.

In many cases classes are used to represent real-world entities (such as employees), but they do not need to, they can also represent more abstract concepts such as a transaction between one person and another (for example, an agreement to buy a meal).

Classes act as *templates* which are used to construct instances or examples of a class of things. Each example of the class `Person` has a name, an age, an address, etc., but they have their own values for their name, age and address. For example, to represent the people in a family we might create a class Person with the name Paul, the age 52 and the address set to London. We may also create another Person object

© Springer Nature Switzerland AG 2023

J. Hunt, *A Beginners Guide to Python 3 Programming*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-35122-8_18

(instance) with the name Fiona, the age 48 and the address also of London and so on.

An instance or object is therefore an example of a class. All instances/objects of a class possess the same data variables but contain their own data. Each instance of a class responds to the same set of requests.

Classes allow programmers to specify the *structure* of an object (i.e., its attributes or fields, etc.) and the its behaviour separately from the objects themselves.

This is important, as it would be extremely time-consuming (as well as inefficient) for programmers to define each object individually. Instead, they define classes and create *instances* or *objects* of those classes.

They can then store related data together in a named concept which makes it much easier to structure and maintain code.

18.3 What Are Classes for?

We have already seen several types of data in Python such as integer, string, Boolean, etc. Each of these allowed us to hold a single item of data (such as the integer 42 or the string 'John' and the value `True`). However, how might we represent a Person, a Student or an Employee of a firm? One way we can do this is to use a class to represent them.

As indicated above, we might represent any type of (more complex) data item use a combination of attributes (or fields) and behaviours. These attributes will use existing data types, these might be integers, strings, Booleans, floating-point numbers or other classes.

For example, when defining the class `Person` we might give it:

- a field or attribute for the person's name,
- a field or attribute for their age,
- a field or attribute for their email,
- some behaviour to give them a birthday (which will increment their age),
- some behaviour to allow us to send them a message via their email,
- etc.

In Python classes are thus used:

- as a template to create instances (or objects) of that class,
- define instance methods or common behaviour for a class of objects,
- define attributes or fields to hold data within the objects,
- be sent messages.

Objects (or instances), on the other hand, can:

- be created from a class,
- hold their own values for instance variables,
- be sent messages,

- execute instance methods,
- may have many copies in the system (all with their own data).

18.3.1 What Should a Class Do?

A class should accomplish one specific purpose; it should capture only one idea. If more than one idea is encapsulated in a class, you may reduce the chances for reuse, as well as contravene the laws of encapsulation in object-oriented systems. For example, you may have merged two concepts together so that one can directly access the data of another. This is rarely desirable.

The following guidelines may help you to decide whether to split the class with which you are working. Look at the comment describing the class (if there is no class comment, this is a bad sign in itself). Consider the following points:

- Is the description of the class short and clear? If not, is this a reflection on the class? Consider how the comment can be broken down into a series of short clear comments. Base the new classes around those comments.
- If the comment is short and clear, do the class and instance variables make sense within the context of the comment? If they do not, then the class needs to be re-evaluated. It may be that the comment is inappropriate, or the class and instance variables inappropriate.
- Look at how and where the attributes of the class are used. Is their use in line with the class comment? If not, then you should take appropriate action.

18.3.2 Class Terminology

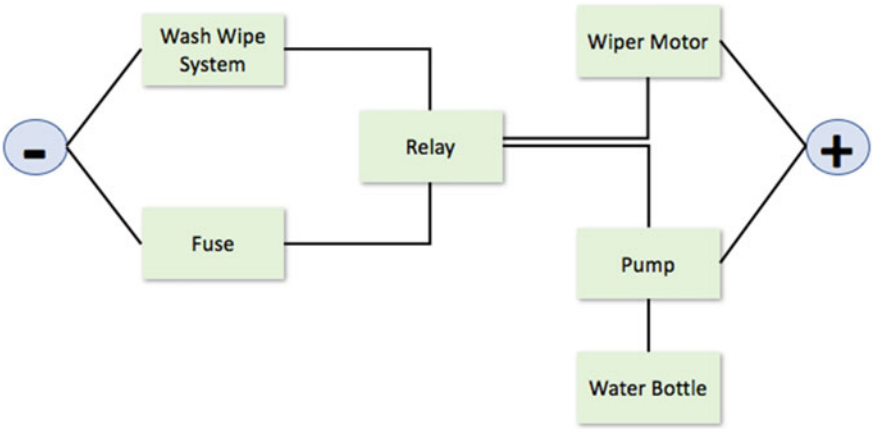
The following terms are used in Python (and other languages that support object orientation):

- *Class* A class defines a combination of data and behaviour that operates on that data. A class acts as a template when creating new instances.
- *Instance or object* An instance also known as an object is an example of a class. All instances of a class possess the same data fields/attributes but contain their own data values. Each instance of a class responds to the same set of requests.
- *Attribute/field/instance variable* The data held by an object is represented by its attributes (also sometimes known as a field or an instance variable). The "state" of an object at any particular moment relates to the current values held by its attributes.
- *Method* A method is a procedure defined within an object.
- *Message* A message is sent to an object requesting some operation to be performed or some attribute to be accessed. It is a request to the object to do something or return something. However, it is up to the object to determine how to execute that request. A message may be considered akin to a procedure call in other languages.

18.4 How is an OO System Constructed?

At this point you may be wondering how a system can be built from classes and objects instantiated from those classes? What would such an application look like? It is clear it is different to writing functions and free-standing application code that calls those functions?

Let's use a real-world (physical) system to explore what an OOP application might look like.



This system aims to provide a diagnosis tutor for the equipment illustrated above. Rather than use the wash–wipe system from a real car, students on a car mechanics diagnosis course use this software simulation. The software system mimics the actual system, so the behaviour of the pump depends on information provided by the relay and the water bottle.

The operation of the wash–wipe system is controlled by a switch which can be in one of five positions: off, intermittent, slow, fast and wash. Each of these settings places the system into a different state:

Switch setting	System state
`Off`	The system is inactive
`Intermittent`	The blades wipe the windscreen every few seconds
`Slow`	The wiper blades wipe the windscreen continuously
`Fast`	The wiper blades wipe the windscreen continuously and quickly
`Wash`	The pump draws water from the water bottle and sprays it onto the windscreen

For the pump and the wiper motor to work correctly, the relay must function correctly. In turn, the relay must be supplied with an electrical circuit. This electrical

circuit is negatively fused, and thus, the fuse must be intact for the circuit to be made. Cars are negatively switched as this reduces the chances of short circuits leading to unintentional switching of circuits.

18.4.1 Where Do We Start?

This is often a very difficult point for those new to object-oriented systems. That is, they have read the basics and understand simple diagrams, but do not know where to start. It is the old chestnut, "I understand the example but don't know how to apply the concepts myself". This is not unusual and, in the case of object orientation, is probably normal.

The answer to the question "where do I start?" may at first seem somewhat obscure; you should start with the data. Remember that objects are things that exchange messages with each other. The things possess the data that is held by the system and the messages request actions that relate to the data. Thus, an object-oriented system is fundamentally concerned with data items.

Before we go on to consider the object-oriented view of the system, let us stop and think for a while. Ask yourself where could I start; it might be that you think about starting "with some form of functional decomposition" (breaking the problem down in terms of the functions it provides) as this might well be the view the user has of the system. As a natural part of this exercise, you would identify the data required to support the desired functionality. Notice that the emphasis would be on the system functionality.

Let us take this further and consider the functions we might identify for the example presented above:

Function	Description
`Wash`	Pump water from the water bottle to the windscreen
`Wipe`	Move the windscreen wipers across the windscreen

We would then identify important system variables and sub-functions to support the above functions.

Now let us go back to the object-oriented view of the world. In this view, we place a great deal more emphasis on the data items involved and consider the operations associated with them (effectively, the reverse of the functional decomposition view). This means that we start by attempting to identify the primary data items in the system; next, we look to see what operations are applied to, or performed on, the data items; finally, we group the data items and operations together to form objects. In identifying the operations, we may well have to consider additional data items, which may be separate objects or attributes of the current object. Identifying them is mostly a matter of skill and experience.

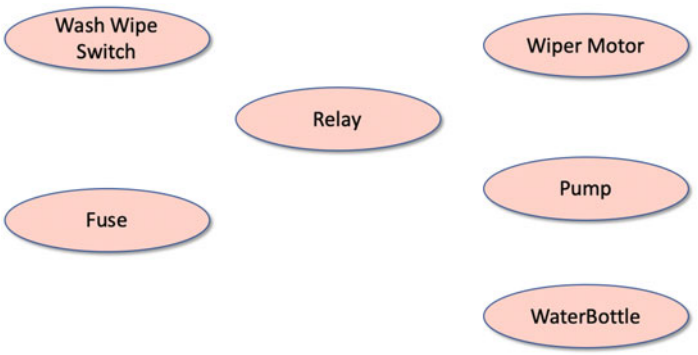
The object-oriented design approach considers the operations far less important than the data and their relationships. In the next section we examine the objects that might exist in our simulation system.

18.4.2 Identifying the Objects

We look at the system as a whole and ask what indicates the state of the system. We might say that the position of the switch or the status of the pump is significant. This results in the data items shown below

Data item	States
`switch setting`	Is the switch set to off, intermittent, wipe, fast wipe or wash?
`wiper motor`	Is the motor working or not?
`pump state`	Is the pump working or not?
`fuse condition`	Has the fuse blown or not?
`water bottle level`	The current water level
`relay status`	Is current flowing or not?

The identification of the data items is considered in greater detail later. At this point, merely notice that we have not yet mentioned the functionality of the system or how it might fit together, we have only mentioned the significant items. As this is such a simple system, we can assume that each of these elements is an object and illustrate it in a simple object diagram:



Notice that we have named each object after the element associated with the data item (e.g., the element associated with the fuse condition is the fuse itself) and that the actual data (e.g., the condition of the fuse) is an instance variable of the object. This is a very common way of naming objects and their instance variables. We now have the basic objects required for our application.

18.4.3 Identifying the Services or Methods

At the moment, we have a set of objects each of which can hold some data. For example, the water bottle can hold an integer indicating the current water level. Although object-oriented systems are structured around the data, we still need some procedural content to change the state of an object or to make the system achieve some goal. Therefore, we also need to consider the operations a user of each object might require. Notice that the emphasis here is on the user of the object and what they require of the object, rather than what operations are performed on the data.

Let us start with the switch object. The switch state can take a number of values. As we do not want other objects to have direct access to this variable, we must identify the services that the switch should offer. As a user of a switch we want to be able to move it between its various settings. As these settings are essentially an enumerated type, we can have the concept of incrementing or decrementing the switch position. A switch must therefore provide a `move_up` and a `move_down` interface. Exactly how this is done depends on the programming language; for now, we concentrate on specifying the required facilities.

If we examine each object in our system and identify the required services, we may end up with the following table:

Object	Service	Description
switch	move_up	Increment switch value
	move_down	Decrement switch value
	State?	Return a value indicating the current switch state
fuse	working?	Indicate if the fuse has blown or not
wiper motor	working?	Indicate whether the wipers are working or not
relay	working?	Indicate whether the relay is active or not
pump	working?	Indicate whether the pump is active or not
water bottle	fill	Fill the water bottle with water
	extract	Remove some water from the water bottle
	empty	Empty the water bottle

We generated this table by examining each of the objects in isolation to identify the services that might reasonably be required. We may well identify further services when we attempt to put it all together.

Each of these services should relate to a method within the object. For example, the moveUp and moveDown services should relate to methods that change the state instance variable within the object. Using a generic pseudo-code, the `move_up` method, within the switch object, might contain the following code:

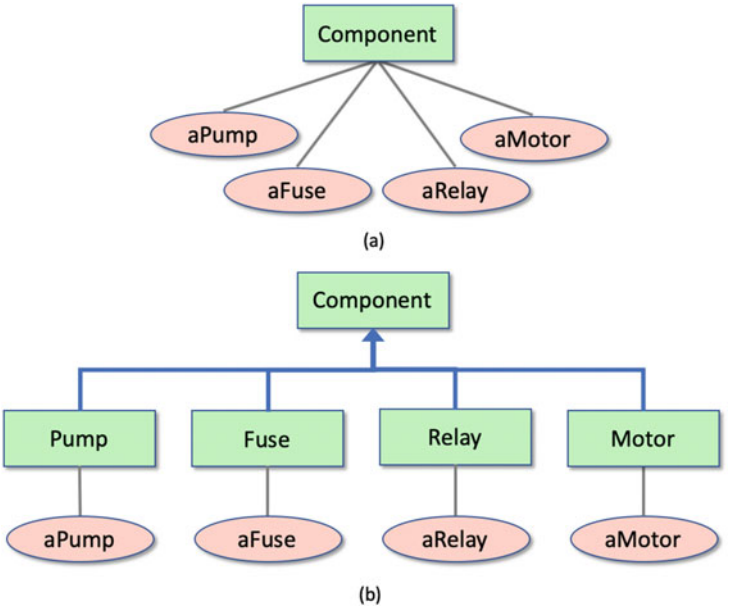
```
def move_up(self):
   if self.state = = "off" then
      self.tate = "wash"
   else if self.state = = "wash" then
```

```
self.state = "wipe"
```

This method changes the value of the state variable in switch. The new value of the instance variable depends on its previous value. You can define moveDown in a similar manner. Notice that the reference to the instance variable illustrates that it is global to the object. The moveUp method requires no parameters. In object-oriented systems, it is common for few parameters to be passed between methods (particularly of the same object), as it is the object that holds the data.

18.4.4 Refining the Objects

If we look back to able table, we can see that fuse, wiper motor, relay and pump all possess a service called `working?`. This is a hint that these objects may have something in common. Each of them presents the same interface to the outside world. If we then consider their attributes, they all possess a common instance variable. At this point, it is too early to say whether fuse, wiper motor, relay and pump are all instances of the same class of object (e.g., a Component class) or whether they are all instances of classes which inherit from some common superclass (see below). However, this is something we must bear in mind later.



18.4.5 Bringing It All Together

So far, we have identified the primary objects in our system and the basic set of services they should present. These services were based solely on the data the objects hold. We must now consider how to make our system function. To do this, we need to consider how it might be used. The system is part of a very simple diagnosis tutor; a student uses the system to learn about the effects of various faults on the operation of a real wiper system, without the need for expensive electronics. We therefore wish to allow a user of the system to carry out the following operations:

- change the state of a component device
- ask the motor what its new state is

The `move_up` and `move_down` operations on the switch change the switch's state. Similar operations can be provided for the fuse, the water bottle and the relay. For the fuse and the relay, we might provide a `change_state` interface using the following algorithm:

```
define change_state(self)
   if self.state = = "working" then
      self.tate = "notWorking"
   else
      self.state = "working"
```

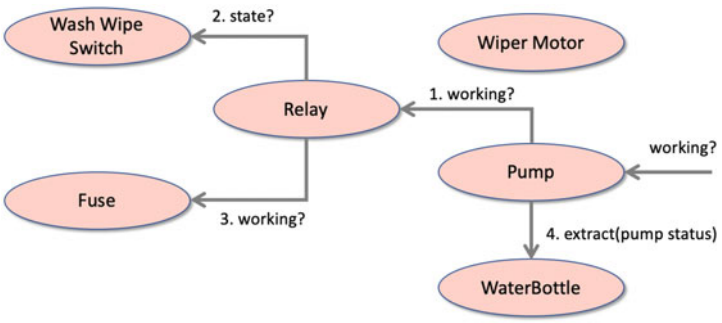
Discovering the state of the motor is more complicated. We have encountered a situation where one object's state (the value of its instance variable) is dependent on information provided by other objects. If we write down procedurally how the value of other objects affect the status of the pump, we might get the following pseudo-code:

```
if fuse is working then
   if switch is not off then
      if relay is working then
         pump status = "working"
```

This algorithm says that the pump status depends on the relay status, the switch setting and the fuse status. This is the sort of algorithm you might expect to find in your application. It links the sub-functions together and processes the data.

In an object-oriented system, well-mannered objects pass messages to one another. How then do we achieve the same effect as the above algorithm? The answer is that we must get the objects to pass messages requesting the appropriate information. One way to do that is to define a method in the pump object that gets the required information from the other objects and determines the motor's state. However, this requires the pump to have links to all the other objects so that it can send them messages. This is a little contrived and loses the structure of the underlying system. It also loses any modularity in the system. That is, if we want to add new components then we have to change the pump object, even if the new components only affect the switch. This approach also indicates that the developer is thinking too procedurally and not really in terms of objects.

In an object-oriented view of the system, the pump object only needs to know the state of the relay. It should therefore request this information from the relay. In turn, the relay must request information from the switches and the fuse.



The above illustrates the chain of messages initiated by the pump object:

1. pump sends a working? message to the relay,
2. relay sends a state? message to the switch, the switch replies to the relay,
3. relay sends a second working? message to the fuse:
4. The fuse replies to the relay
5. the relay replies to the motor
6. If the pump is working, then the pump object sends the final message to the water bottle
7. pump sends a message extract to the water bottle

In step four, a parameter is passed with the message because, unlike the previous messages that merely requested state information, this message requests a change in state. The parameter indicates the rate at which the pump draws water from the water bottle.

The water bottle should not record the value of the pump's status as it does not own this value. If it needs the motor's status in the future, it should request it from the pump rather than using the (potentially obsolete) value passed to it previously.

In the above figure we assumed that the pump provided the service `working?` which allows the process to start. For completeness, the pseudo-code of the `working?` method for the pump object is:

```
def working?(self)
   self.status = relay.working()
   if self.status = = "working" then
      water_bottle.extract(self.status)
```

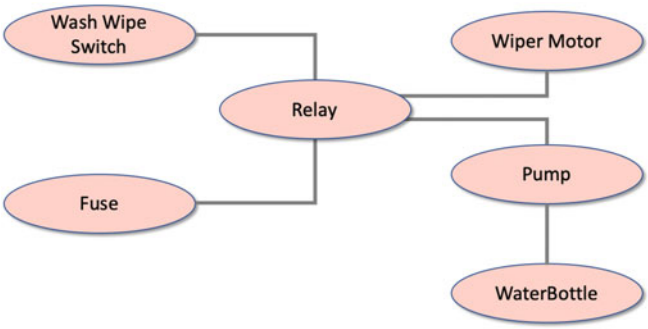
This method is a lot simpler than the procedural program presented earlier. At no point do we change the value of any variables that are not part of the pump, although they may have been changed as a result of the messages being sent. Also, it only shows us the part of the story that is directly relevant to the pump. This means that it can

be much more difficult to deduce the operation of an object-oriented system merely by reading the source code. Some Python environments (such as the PyCharm IDE) alleviate this problem, to some extent, through the use of sophisticated browsers.

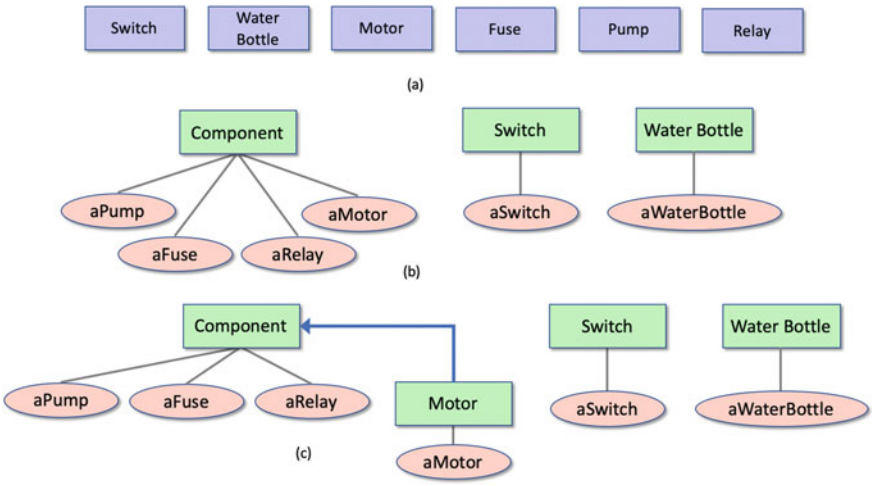
18.5 Where Is the Structure in an OO Program?

People new to object orientation may be confused because they have lost one of the key elements that they use to help them understand and structure a software system: the main program body. This is because the objects and the interactions between them are the cornerstone of the system. In many ways, the following figure shows the object-oriented equivalent of a main program. This also highlights an important feature of most object-oriented approaches: graphical illustrations. Many aspects of object technology, for example object structure, class inheritance and message chains, are most easily explained graphically.

Let us now consider the structure of our object-oriented system. It is dictated by the messages that are sent between objects. That is, an object must possess a reference to another object in order to send it a message. The resulting system structure is illustrated below.



In Python, this structure is achieved by making instance variables reference the appropriate objects. This is the structure which exists between the instances in the system and does not relate to the classes, which act as templates for the instances.

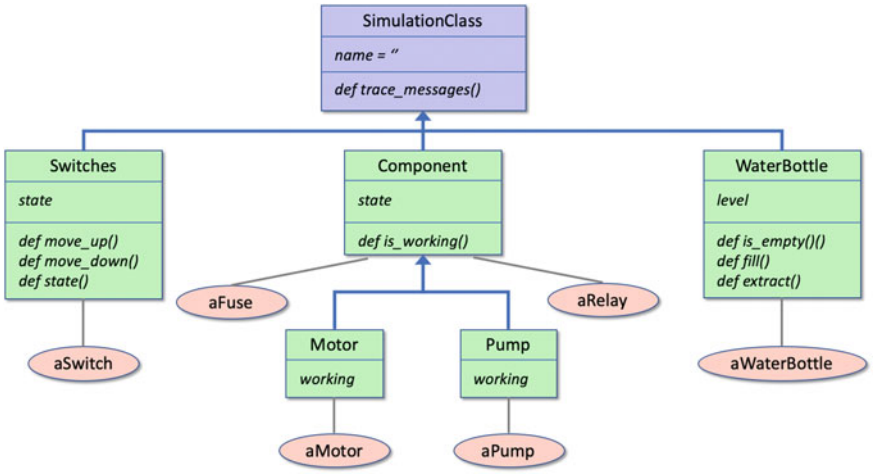


We now consider the classes that create the instances. We could assume that each object is an instance of an equivalent class (see above (a)). However, as has already been noted, some of the classes bear a very strong resemblance. In particular, the fuse, the relay, the motor and the pump share a number of common features. Table following table compares the features (instance variables and services) of these objects.

	fuse	relay	motor	pump
instance variable	state	state	state	state
services	working?	working?	working?	working?

From this table, the objects differ only in name. This suggests that they are all instances of a common class such as Component. This class would possess an additional instance variable, to simplify object identification.

If they are all instances of a common class, they must all behave in exactly the same way. However, we want the pump to start the analysis process when it receives the message working? so it must possess a different definition of working? from fuse and relay. In other ways it is very similar to fuse and relay, so they can be instances of a class (say Component) and pump and motor can be instances of classes that inherit from Component (but redefine working?). This is illustrated in the previous figure (c). The full class diagram is presented in the Figure below.



18.6 Further Readings

If you want to explore some of the ideas presented in this chapter in more detail here are some online references:

- https://en.wikipedia.org/wiki/Object-oriented_programming This is the Wikipedia entry for Object-Oriented Programming and thus provides a quick reference to much of the terminology and history of the subject and acts as a jumping off point for other references.
- https://dev.to/charanrajgolla/beginners-guide---object-oriented-programming which provides a light-hearted look at the four concepts within object orientations namely abstraction, inheritance, polymorphism and encapsulation.
- https://www.tutorialspoint.com/python/python_classes_objects.htm A Tutorials Point course on Object-Oriented Programming and Python.