Inheritance, Encapsulation and Polymorphism

We have already seen the modeling power of OOP using the class and object functions by combining data and methods. There are three more important concept, **inheritance**, which makes the OOP code more modular, easier to reuse and build a relationship between classes. **Encapsulation** can hide some of the private details of a class from other objects, while **polymorphism** can allow us to use a common operation in different ways. In this section, we will briefly discuss them.

Inheritance

Inheritance allows us to define a class that inherits all the methods and attributes from another class. Convention denotes the new class as **child class**, and the one that it inherits from is called **parent class** or **superclass**. If we refer back to the definition of class structure, we can see the structure for basic inheritance is **class ClassName(superclass)**, which means the new class can access all the attributes and methods from the superclass. Inheritance builds a relationship between the child class and parent class, usually in a way that the parent class is a general type while the child class is a specific type. Let us try to see an example.

TRY IT! Define a class named `Sensor` with attributes `name`, `location`, and `record_date` that pass from the creation of an object and an attribute `data` as an empty dictionary to store data. Create one method *add_data* with `t` and `data` as input parameters to take in timestamp and data arrays. Within this method, assign `t` and `data` to the `data` attribute with 'time' and 'data' as the keys. In addition, it should have one `clear_data` method to delete the data.

```python
class Sensor():
    def __init__(self, name, location, record_date):
        self.name = name
        self.location = location
        self.record_date = record_date
        self.data = {}

    def add_data(self, t, data):
        self.data['time'] = t
        self.data['data'] = data
        print(f'We have {len(data)} points saved')

    def clear_data(self):
        self.data = {}
        print('Data cleared!')
```

Now we have a class to store general sensor information, we can create a sensor object to store some data.

EXAMPLE: Create a sensor object.

```
import numpy as np

sensor1 = Sensor('sensor1', 'Berkeley', '2019-01-01')
data = np.random.randint(-10, 10, 10)
sensor1.add_data(np.arange(10), data)
sensor1.data
```

```
We have 10 points saved
```

```
{'time': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
 'data': array([-4, -7,  2, -3, -8,  6,  4,  3,  5, -9])}
```

Inherit and extend new method

Say we have one different type of sensor: an accelerometer. It shares the same attributes and methods as `Sensor` class, but it also has different attributes or methods need to be appended or modified from the original class. What should we do? Do we create a different class from scratch? This is where inheritance can be used to make life easier. This new class will inherit from the `Sensor` class with all the attributes and methods. We can whether we want to extend the attributes or methods. Let us first create this new class, `Accelerometer`, and add a new method, `show_type`, to report what kind of sensor it is.

```
class Accelerometer(Sensor):

    def show_type(self):
        print('I am an accelerometer!')

acc = Accelerometer('acc1', 'Oakland', '2019-02-01')
acc.show_type()
data = np.random.randint(-10, 10, 10)
acc.add_data(np.arange(10), data)
acc.data
```

```
I am an accelerometer!
We have 10 points saved
```

```
{'time': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
 'data': array([ -2,   2, -10,   6,   2,  -8,   2,   3,   7,  -6])}
```

Creating this new `Accelerometer` class is very simple. We inherit from `Sensor` (denoted as a superclass), and the new class actually contains all the attributes and methods from the superclass. We then add a new method, `show_type`, which does not exist in the `Sensor` class, but we can successfully extend the child class by adding the new method. This shows the power of inheritance: we have reused most part of the `Sensor` class in a new class, and extended the functionality. Besides, the inheritance sets up a logical relationship for the modeling of the real-world entities : the `Sensor` class as the parent class is more general and passes all the characteristics to the child class `Accelerometer`.

Inherit and method overriding

When we inherit from a parent class, we can change the implementation of a method provided by the parent class, this is called method overriding. Let us see the following example.

EXAMPLE: Create a class `UCBAcc` (a specific type of accelerometer that created at UC Berkeley) that inherits from `Accelerometer` but replace the `show_type` method that prints out the name of the sensor.

```python
class UCBAcc(Accelerometer):

    def show_type(self):
        print(f'I am {self.name}, created at UC Berkeley!')

acc_ucb = UCBAcc('UCBAcc', 'Berkeley', '2019-03-01')
acc_ucb.show_type()
```

```
I am UCBAcc, created at UC Berkeley!
```

We see that, our new `UCBAcc` class actually overrides the method `show_type` with new features. In this example, we are not only inheriting features from our parent class, but we are also modifying/improving some methods.

Inherit and update attributes with super

Let us create a class `NewSensor` that inherits from `Sensor` class, but with updated the attributes by adding a new attribute `brand`. Of course, we can re-define the whole `__init__` method as shown below and overriding the parent function.

```python
class NewSensor(Sensor):
    def __init__(self, name, location, record_date, brand):
        self.name = name
        self.location = location
        self.record_date = record_date
        self.brand = brand
        self.data = {}
```

```
new_sensor = NewSensor('OK', 'SF', '2019-03-01', 'XYZ')
new_sensor.brand
```

```
'XYZ'
```

However, there is a better way to achieve the same. We can use the `super` method to avoid referring to the parent class explicitly. Let us see how to perform this in the following example:

EXAMPLE: Redefine the attributes in inheritance.

```
class NewSensor(Sensor):
    def __init__(self, name, location, record_date, brand):
        super().__init__(name, location, record_date)
        self.brand = brand

new_sensor = NewSensor('OK', 'SF', '2019-03-01', 'XYZ')
new_sensor.brand
```

```
'XYZ'
```

Now we can see with the *super* method, we avoid to list all the definition of the attributes, this helps keep your code maintainable for the foreseeable future. But it really useful when you are doing multiple inheritance, which is beyond the discussion of this book.

Encapsulation

Encapsulation is one of the fundamental concepts in OOP. It describes the idea of restricting access to methods and attributes in a class. This will hide the complex details from the users, and prevent data being modified by accident. In Python, this is achieved by using private methods or attributes using underscore as prefix, i.e. single "_" or double "__". Let us see the following example.

EXAMPLE:

```
class Sensor():
    def __init__(self, name, location):
        self.name = name
        self._location = location
        self.__version = '1.0'

    # a getter function
    def get_version(self):
        print(f'The sensor version is {self.__version}')

    # a setter function
    def set_version(self, version):
        self.__version = version
```

```python
sensor1 = Sensor('Acc', 'Berkeley')
print(sensor1.name)
print(sensor1._location)
print(sensor1.__version)
```

```
Acc
Berkeley
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-8-ca9b481690ba> in <module>
      2 print(sensor1.name)
      3 print(sensor1._location)
----> 4 print(sensor1.__version)


AttributeError: 'Sensor' object has no attribute '__version'
```

The above example shows how the encapsulation works. With single underscore, we defined a private variable, and it should not be accessed directly. But this is just convention, nothing stops you from doing that. You can still get access to it if you want to. With double underscore, we can see that the attribute `__version` can not be accessed or modify it directly. Therefore, to get access to the double underscore attributes, we need to use getter and setter function to access it internally, as shown in the following example.

```python
sensor1.get_version()
```

```
The sensor version is 1.0
```

```python
sensor1.set_version('2.0')
sensor1.get_version()
```

```
The sensor version is 2.0
```

The single and double underscore also apply to private methods as well, we will not discuss these as they are similar to the private attributes.

Polymorphism

Polymorphism is another fundamental concept in OOP, which means multiple forms. Polymorphism allows us to use a single interface with different underlying forms such as data types or classes. For example, we can have commonly named methods across classes or child classes. We have already seen one example above, when we override the method `show_type` in the `UCBAcc`. For parent class `Accelerometer` and child class `UCBAcc`, they both have a method named `show_type`, but they have different implementation. This ability of using single name with many forms acting differently in different situations greatly reduces our complexities. We will not expand to discuss more of Polymorphism, if you are interested, check more online to get a deeper understanding.

< 7.2 Class and Object | Contents | 7.4 Summary and Problems >