Modelling things using Python classes

Estimated Time: 1-2 hours

Python's classes allow us to model complicated, real-life things in code. Until now we have used simple data types to model simple things. For example you could use a number to model the price of a tin of Spam, or a Boolean value to model whether a user likes anchovies on their pizza.

When we need to model more complicated things we can use classes. For example, let's consider Nigel Tufnel, lead guitarist of the popular beat combo Spiñal Tap. How could we represent Nigel in a program?

First we can think about a musician. All musicians have certain characteristics which make them musicians rather than, say, electricians or lawyers. For example, they have an instrument and can play solos.

Now think about Nigel. He is a specific musician, playing all of the classic Tap hits. There are other instances of musicians out there, Lionel Richie for instance, but they are all fundamentally musicians.

In Python Nigel would be an **instance** of a Musician **class**. The class represents the idea of a musician, whereas an instance represents an actual musician.

Let's see how this looks in code. First let's define a basic Musician class:

```
class Musician(object):
     pass
```

Here we create the `Musician` class and tell it to inherit from `object`. We will learn more about inheritance in a moment, so don't worry too much about this yet. Finally we `pass`, which tells the code to do nothing (for now).

We can use this class to make instance of musicians:

```
>>> david = Musician()
>>> derek = Musician()
>>> david == derek
False
```

Notice how `david` and `derek` are different musicians.

At the moment our Musicians aren't very musical. So let's teach them some sounds by adding attributes to the instance:

```
>>> david.sounds = ["Twang", "Thrumb", "Bling"]
>>> derek.sounds = ["Boink", "Bow", "Boom"]
```

You can add any attributes to any instance. But when you know that instances of a class should always contain an attribute we can use a shortcut: the `__init__` method:

```
class Musician(object):
    def __init__(self, sounds):
        self.sounds = sounds
```

The `__init__` method is automatically called whenever you create a Musician instance. Any arguments which you pass into `Musician()` will be passed to `__init__()`. For example:

```
>>> david = Musician(["Twang", "Thrumb", "Bling"])
>>> derek = Musician(["Boink", "Bow", "Boom"])
```

But wait a minute! We are only passing one argument into the function, and `__init__` has two parameters. This is because for any **method** (i.e. any function inside a class) Python automatically puts the instance into the `self` variable. So when we say `self.sounds` this is the same as saying `david.sounds`, or `derek.sounds`.

We can also add other methods to a Python class to give it behavior. For example to add the soloing behavior:

```
class Musician(object):
    def __init__(self, sounds):
        self.sounds = sounds

    def solo(self, length):
        for i in range(length):
            print(self.sounds[i % len(self.sounds)], end=" ")
        print()
```

Then to play a solo:

```
>>> david = Musician(["Twang", "Thrumb", "Bling"])
>>> david.solo(6)
```

Now if we are going to have a whole band we are going to need more than one type of musician. And each musician is going to need different behaviors. For example a guitarist will need to tune their guitar, whereas a bassist is unlikely to know how. Well, we could create two classes to represent bassists and guitarists:

```
class Bassist(object):
    def __init__(self):
        self.sounds = ["Twang", "Thrumb", "Bling"]

    def solo(self, length):
        for i in range(length):
            print(self.sounds[i % len(self.sounds)], end=" ")
        print()

class Guitarist(object):
    def __init__(self):
        self.sounds = ["Boink", "Bow", "Boom"]


    def solo(self, length):
        for i in range(length):
            print(self.sounds[i % len(self.sounds)], end=" ")
        print()

    def tune(self):
        print("Be with you in a moment")
        print("Twoning, sproing, splang")
```

That seems pretty tedious though. Wouldn't it be great if we could share the common behaviors and attributes? Luckily this is where **inheritance** comes in:

```
class Musician(object):
    def __init__(self, sounds):
        self.sounds = sounds

    def solo(self, length):
        for i in range(length):
            print(self.sounds[i % len(self.sounds)], end=" ")
        print()

class Bassist(Musician): # The Musician class is the parent of the Bas
    def __init__(self):
        # Call the __init__ method of the parent class
        super().__init__(["Twang", "Thrumb", "Bling"])

class Guitarist(Musician):
    def __init__(self):
        # Call the __init__ method of the parent class
        super().__init__(["Boink", "Bow", "Boom"])
    def tune(self):
        print("Be with you in a moment")
        print("Twoning, sproing, splang")
```

Here the `Bassist` and `Guitarist` classes inherit the attributes and behaviors of the `Musician` class. The inherited attributes and methods can be used as usual:

```
nigel = Guitarist()
nigel.solo(6)
print(nigel.sounds)
```

Also notice how the child classes define their own `__init__` methods which override the parent `__init__` method. Any method of a parent can be overridden by its child. So for example a `ProgRockGuitarist` could have a more convoluted `solo` method. The parent methods aren't lost completely though – we can use the `super` method to gain access to them. For example in the child `__init__` methods we use the `super` function to call the parent `__init__` methods.

Try It!

Extend the example code to add a `Drummer` class. Drummers should be able to solo, count to four, and spontaneously combust. Then add a `Band` class. Bands should be able to hire and fire musicians, and have the musicians play their solos after the drummer has counted time.

☆ ☆ ☆ ☆ ☆ · Report a typo or other issue

https://gist.github.com/ray-newby/3fa2bc74

Completed

Previous

Next